A GUIDE FOR MANAGERS

The Economics of Flow: A Guide for Managers

Why waiting costs more than working, and what to do about it

Contents

How to read this guide 3

PART I SEEING THE ECONOMICS

0 Why your numbers don't show the money 7

1 What flow is 11

2 Time is money: the cost of delay 17

3 Throughput money: where the system makes money 24

PART II WHY WORK WAITS

4 Queues: the inventory nobody sees 30

5 Utilisation and slack: why "everyone is busy" is expensive 35

6 Variability, uncertainty and fat tails 40

PART III LEVERS

7 Constraints: where to focus 47

8 Batch size: the cheapest lever 51

9 WIP limits, aging and flow debt 56

10 What goes first: sequencing and classes of service 61

11 Promises and forecasts as economic tools 65

12 Cadence and feedback 70

PART IV MANAGING THE SYSTEM

13 Metrics that show the economics 74

14 Management as the constraint 78

15 Cheat sheet, where to start, glossary 83

Team Atlas: the whole case in one place 87

Sources 91

How to read this guide

Why this guide exists

Donald Reinertsen, who spent around 30 years advising product development organisations, makes an uncomfortable observation: most managers do not understand the economics of their own products. Ask the people on one project what a 60-day delay would cost, and their answers differ by a factor of fifty.

Without that understanding, everyday decisions about time, capacity, priorities and quality are made on intuition. Many of them lose money, and nobody notices, because the losses never appear in any report.

This guide tries to close that gap. It explains what **flow** is, how work actually moves through an organisation, why it waits, what the waiting costs, and which few numbers make all of this visible.

Four ways of looking at the same problem

No single author has the whole picture. This guide draws on four schools of thought. Each one answers a different question a manager might ask.

LENS	THE QUESTION IT ANSWERS	MAIN SOURCES
The economics of time	What does delay cost, and when is it worth paying to go faster?	Donald Reinertsen
Constraints and throughput money	Where does the system actually make money, and what limits it?	Eliyahu Goldratt, David Anderson, Steve Tendon
Predictability from data	How long does work really take, and what can we honestly promise?	Daniel Vacanti, Troy Magennis, Alexei Zheglov
Service and risk	Which work is urgent, which can wait, and how do we manage the risk?	The *Kanban Maturity Model* (David Anderson and Teodora Bozheva)

Most of the time these schools agree, and where they do, you can be confident. Sometimes they disagree. When that happens, this guide shows both views, explains when each one holds and gives you a practical choice. None of them is "the answer". Each is a lens, and good managers learn to switch between them.

How the guide is built

Each chapter starts with an everyday situation, explains new terms the first time they appear, and ends with a short "**What this means for you**" box. You don't need a background in economics, statistics or Kanban.

THREE WORDS YOU WILL MEET.

Kanban, or more precisely the Kanban Method, "means applying a holistic way of thinking about your services with a focus on improving them from your customers' perspective" (David Anderson); the *Kanban Maturity Model* (KMM) describes how organisations grow into it. **TameFlow** is Steve Tendon's approach that applies Goldratt's Theory of Constraints to knowledge work. A **service level expectation (SLE)** is a promise from data, such as "85% of items within 16 days"; Chapter 11 covers it in full.

PART	CHAPTERS	WHAT IT COVERS
I. Seeing the economics	0–3	Why normal management numbers hide the money; what flow is; the cost of delay; throughput money
II. Why work waits	4–6	Queues, utilisation and slack, variability
III. Levers	7–12	Constraints, batch size, WIP limits, sequencing, promises and forecasts, cadence and feedback
IV. Managing the system	13–15	Metrics, management as the constraint, where to start

Five terms used throughout

TERM	MEANING IN THIS GUIDE
Committed	The moment the team agrees to do an item. Before it, an idea waits in a list: you have paid to acquire it (Anderson counts this as investment), but waiting costs almost nothing. After it, the clock runs
Lead Time	Calendar days from Committed until the item is ready for delivery, waiting included
WIP (work in progress)	Everything committed and not yet ready for delivery, whether someone is working on it or it is waiting
Delivery rate	How many items reach ready for delivery per week, the same boundary as Lead Time and WIP, so Little's Law holds. (Chapter 3 explains why we avoid calling this "throughput")
Percentile	"X% of items finished within this many days". The 85th percentile of 16 days means 85 out of 100 items were ready for delivery within 16 days

A NOTE ON WORDS.

Authors use different names for the same thing. Reinertsen says **cycle time** for the total time work takes to get through a process, waiting included; Vacanti uses **cycle time** for time inside a defined workflow; the *Kanban Maturity Model* says **Lead Time**. In this guide we use **Lead Time** as defined above. Quotes keep the author's own word: where Reinertsen says *cycle time*, read Lead Time. Alexei Zheglov goes further: there is no such thing as "the cycle time". In manufacturing the term means the interval between deliveries, so whenever you hear it, ask where the clock starts and stops.

A note on money. Short worked examples are in euros; where an author used dollars and we converted, the text says so. Amounts from published cases and quotes (Goldratt's plant, Magennis' projects, Reinertsen's larger examples, the Boeing rule) stay in the author's **original dollars**, so you can check them against the source.

Meet Team Atlas

TEAM ATLAS IS INVENTED.

It is a fictional team created for this guide. Its weekly delivery and commitment counts, Lead Times, percentiles and forecasts come from a simulated data set; its cost of delay, margin and running cost are illustrative. The numbers are made to behave like a real team's, so the methods work the same way on your own data. The whole case is collected in the section *Team Atlas: the whole case in one place* at the end of the guide.

Over 12 weeks this many items reached ready for delivery each week:

```
Week:     1  2  3  4  5  6  7  8  9 10 11 12
Ready:    3  6  4  0  5  7  2  5  4  6  3  8
```

That's **53 items**, about **4.4 a week**. Week 4 was a production incident: nothing was delivered. On an average day Atlas had about **6 items** in progress. Its median Lead Time was **9 days**, and **85%** of items were ready for delivery within **16 days**. It has **40 items** left in its current release; a forecast from its delivery history gives an 85% chance of finishing within **11 weeks**.

PART I

Seeing the economics

CHAPTER 0

Why your numbers don't show the money

A test you can run on Monday

Reinertsen describes a simple test. Gather ten people who work on the same project: a project manager, a couple of engineers, someone from product, someone from finance. Ask each of them, separately, one question:

> **"If this project is delivered 60 days late, how much profit do we lose?"**

Write down the answers. Then compare the highest with the lowest.

Reinertsen reports that the typical range is **50 to 1**. One person says €20,000. Another says €1,000,000. Same project, same customer, same company. He calls answers like these "quantified ignorance": people produce numbers, but nobody has ever worked them out.

Every week, these same people make decisions that trade time against money. Add a contractor to finish two weeks earlier? Release now and fix in production, or wait for the full test cycle? Start the new request now, or finish what's already open? Keep the specialist fully booked, or leave some slack? If one person thinks a week of delay costs €300 and another thinks it costs €15,000, they will make opposite decisions on every one of these questions, and both will feel sensible.

The robots that made the plant poorer

Eliyahu Goldratt's novel *The Goal* opens with a plant manager, Alex Rogo, whose factory is losing money. He has just installed robots in one department. The robots raised that department's productivity by **36%**. Everyone is proud of the number.

A physicist called Jonah asks Alex two questions. Did the plant ship more products? No. Did inventory go down? No; in fact, it went up. So what did the 36% achieve? Nothing that made money. The robots made parts faster than the rest of the plant could use them. The extra parts sat in piles.

Jonah's point is that **productivity means nothing without a goal**. An action is productive only if it moves the whole company towards making money. A department that gets faster while the company doesn't is a **local optimum**: it looks good in its own numbers and does nothing, or harm, for the whole.

Goldratt's sequel, *It's Not Luck*, adds a second twist. When a company cuts its inventory sharply, which frees cash and shortens delivery times, its accounts can show a **paper loss**, in his example about **$10 million**, because inventory had been valued as an asset. A real improvement looks like a failure in the report. Managers who are judged by that report learn not to improve.

Proxies: the numbers we measure instead

Reinertsen gives the general version of the robot story. Organisations rarely measure the thing they care about, which is profit over the whole life of a product or service (**life-cycle profit**). They measure **proxy variables**: numbers that stand in for it.

PROXY VARIABLE	WHAT IT SEEMS TO TELL US	WHAT IT DOESN'T TELL US
Utilisation ("everyone is busy")	We are using our people well	How much the queues caused by full utilisation cost
Local productivity ("department output +36%")	That part is improving	Whether the whole system makes more money
Lead Time ("we deliver in 16 days")	We are fast	How much a day faster would be worth
Conformance to plan ("on schedule")	We are in control	Whether the plan is still the best economic choice
Cost per unit	We are efficient	Whether lower unit cost came from bigger batches and more waiting

None of these is wrong to measure. The problem is that **they pull against each other, and there is no common unit to compare them**. Push utilisation up, and efficiency improves while Lead Time gets worse. Is that a good trade? You can't say, because one is a percentage and the other is days.

David Anderson, who applied Goldratt's ideas to software development, puts it bluntly: *the goal is profit, not working code*. Working code is a means. If managers can't connect their numbers to money, they optimise the means.

The *Kanban Maturity Model* describes a pattern it calls a trap at the middle level of maturity: organisations that have learned to keep customers happy, but are **losing money doing it**, because nobody connected service delivery to the economics of the business.

The inventory nobody can see

In a factory, unfinished work is visible. Half-built products pile up between machines, take floor space and appear on the balance sheet.

In knowledge work, unfinished work is invisible. Reinertsen calls it **design-in-process inventory**; this guide calls it **WIP**. He gives two reasons nobody sees it:

1. **It is financially invisible.** Development costs are expensed as they occur. If you ask the chief financial officer how much inventory sits in development, the answer will be "Zero".
2. **It is physically invisible.** The work is information: documents, code, designs, test results. In a Hewlett-Packard manager's words, which Reinertsen quotes: "*Our inventory is bits on a disk drive, and we have very big disk drives.*"

Anderson treats this invisible inventory as real money. In his version of Goldratt's accounting (Chapter 3), the requirements and ideas a team has accepted but not delivered are an **investment**: money already spent, waiting to earn something. And like fresh produce, they **go stale**. Markets move, people forget the details, the original requesters leave. A requirement that sits long enough can lose all its value, and the money spent acquiring it becomes a pure loss.

Because the inventory is invisible, so are the **queues** it forms: work waiting for a review, a test environment, a decision, a specialist. Reinertsen calls queues "*the single most important cause of poor product development performance.*" Part II of this guide is about them.

Beliefs that reinforce each other

Reinertsen argues that the usual management habits are not separate mistakes. They form a **self-reinforcing system**. He lists twelve; here are the most important, with the chapters of this guide that deal with them:

THE BELIEF OR HABIT	COVERED IN
Not putting a price on time	Chapters 2 and 3
Blindness to queues	Chapter 4
Keeping everyone busy	Chapter 5
Treating all variability as bad, and the plan as sacred	Chapter 6
Large batches as the norm	Chapter 8
No limits on work in progress	Chapter 9
Prioritising without economics	Chapter 10
Centralised control	Chapter 14

Here is how two of them lock together. Believe in **keeping everyone busy**, and an idle engineer looks like waste. Add **blindness to queues**, and the queues that form when everyone is fully loaded seem free. Put them together, and you load people as close to 100% as you can. Reinertsen reports that executives in his courses described their organisations as running at **98.5% utilisation**. As Chapter 5 shows, that level of loading guarantees long queues and long Lead Times, and nobody connects the two.

Goldratt makes the same point from another angle in *Critical Chain*: *"Tell me how you measure me, and I'll tell you how I'll behave."* If people are measured on local efficiency, they will produce local efficiency, whatever it does to the whole.

How common is this?

Reinertsen gives figures from surveys of people attending his courses. They are his own data, collected before 2009, so treat them as an illustration of a pattern rather than a current benchmark:

PRACTICE	SHARE WHO DO IT
Know the cost of delay of their projects	**15%**
Measure queues	**2%**
Use limits on work in progress	**3%**
Actively try to reduce batch sizes	**3%**

WHAT THIS MEANS FOR YOU

- Most management numbers are **proxies**. They're useful, but they can't tell you whether a trade-off makes or loses money.
- A part of the system can improve while the whole gets worse. Watch for **local optima**.
- Your largest inventory is **invisible**: unfinished work waiting in queues. It goes stale, and nobody prices it.
- Some real improvements **look like losses** in the usual reports.
- **How people are measured is how they behave.**
- The way out starts with two numbers: what **time** is worth (Chapter 2) and where the system **makes money** (Chapter 3). First, though, we need to understand what flow is.

CHAPTER 1

What flow is

Eighty-one hours

In 1926, Henry Ford's River Rouge plant could turn iron ore into a finished car, made of more than 5,000 parts, standing on a railway wagon ready to ship, in **81 hours**.

How? Most people credit only the moving assembly line, though Ford's real breakthrough was continuous flow: eliminating piles of waiting inventory. Decades later, Toyota took this principle further without a fixed-pace conveyor line. By limiting the space for unfinished parts between workstations, they built a pull system with one simple rule: when the space is full, upstream workers stop. They don't keep producing; they pause or help clear the bottleneck.

In other words, both Ford and Toyota deliberately accepted that some people would sometimes be idle, so that work would keep moving.

That is the core of flow.

What "flow" means

Vacanti defines **flow** as *"the movement and delivery of customer value through a process"*. Lead Time measures how long that takes; flow is about **how** the work moves. Good flow means work moves steadily and quickly, with little waiting. Poor flow means work starts, stops, waits, gets reshuffled and finishes late.

Daniel Vacanti, whose books are the standard reference on flow metrics, describes flow with four basic measures. You'll meet them throughout this guide:

MEASURE	WHAT IT COUNTS
WIP	How many items are in progress right now
Lead Time	How long each item took, from Committed to ready for delivery
Delivery rate	How many items reach ready for delivery per week
Work item age	How long each item still in progress has been open so far

The first three are linked by one simple law.

Little's Law

Little's Law, proved by John Little at MIT in 1961, says:

```
Average Lead Time = Average WIP ÷ Average delivery rate
```

If a team finishes about **5 items a week** and has about **20 items** in progress, each item takes on average about **4 weeks**. If WIP rises to **40** while the team still finishes 5 a week, the average becomes **8 weeks**. Nobody worked more slowly. There was simply more work in front of each item.

TEAM ATLAS CHECK. Atlas averaged about **6.2 items** in progress and delivered about **4.4 a week**. Little's Law gives 6.2 ÷ 4.4 ≈ 1.4 weeks, or about **9.9 days**. Atlas's actual average Lead Time, from its item records, was also **9.9 days**.

Goldratt put the same idea in one short line in *Critical Chain*: **"WIP and lead time are twins."** Change one and the other follows.

Vacanti adds an important warning. Little's Law is exact only when certain conditions hold: work arrives at about the rate it leaves, everything that is started gets finished, WIP stays roughly stable, and the average age of work in progress doesn't keep rising. When those conditions break, for example when new work keeps being pushed in or items are quietly abandoned, the averages stop lining up. That is itself useful: **a system where Little's Law doesn't fit is a system that isn't under control.**

Where the time goes

David Anderson breaks an item's Lead Time into four parts:

PART	WHAT HAPPENS
Queue	The item waits to **start** a step because the person or resource is busy. *Example: a change sits in "Ready for review" because both reviewers are on other work*
Setup	Someone gets ready: reads in, sets up an environment, regains context
Process	Someone actually works on it
Wait	The item is **finished** at one step but can't move on yet, often because of batching or release rules. *Example: code is reviewed on Monday but goes to the test environment only with Friday's weekly deployment*

Only **process** adds value. In Anderson's experience, **more than half** of Lead Time is not process time. That is a minimum; other authors give more precise figures. Vacanti reports that teams starting to manage flow typically have a **flow efficiency** (process time ÷ Lead Time) around **15%**. Troy Magennis gives a range of **5–15%**. In plain terms: an item that took 20 days was probably worked on for one to three of them.

This changes where improvement lives. Making the work itself faster is hard. Cutting the waiting is usually far easier and cheaper.

Zheglov gives an everyday benchmark. At a passport office on a busy summer day, his visit took **34 minutes**, of which about **8 minutes** was actual work: a flow efficiency of about 23.5%. He calls that good, and notes that it puts most software organisations to shame. The office also built quality in: the check-in desk checked every form and fixed missing details on the spot, instead of passing defects downstream.

Dependency plus variation

Why does work wait so much, even when everyone works hard? Goldratt answers with a game.

In *The Goal*, Alex Rogo, on a hike with his son's scout troop, sets up a game with five boys, a die and a pile of matchsticks. Each boy, in turn, rolls the die and passes that many matches (up to what he has) to the next boy. The die averages 3.5, so on paper the line should move 3.5 matches per round. Every station has exactly the same capacity. It is a perfectly balanced system.

It ships **noticeably less** than the plan said: in our simulated run, 56 matches against a plan of 70.

The reason is the combination of two ordinary facts:

1. **Dependency.** Each boy can only pass on what the boy before him has given him.
2. **Variation.** Each roll is a bit random.

A boy who rolls high can't pass on more than he has, so his good luck is wasted. A boy who rolls low holds up everyone behind him, so his bad luck is passed on. Over time, **delays accumulate and early finishes are lost**. Inventory builds up in waves in front of the stations that happen to be slow. Nobody did anything wrong.

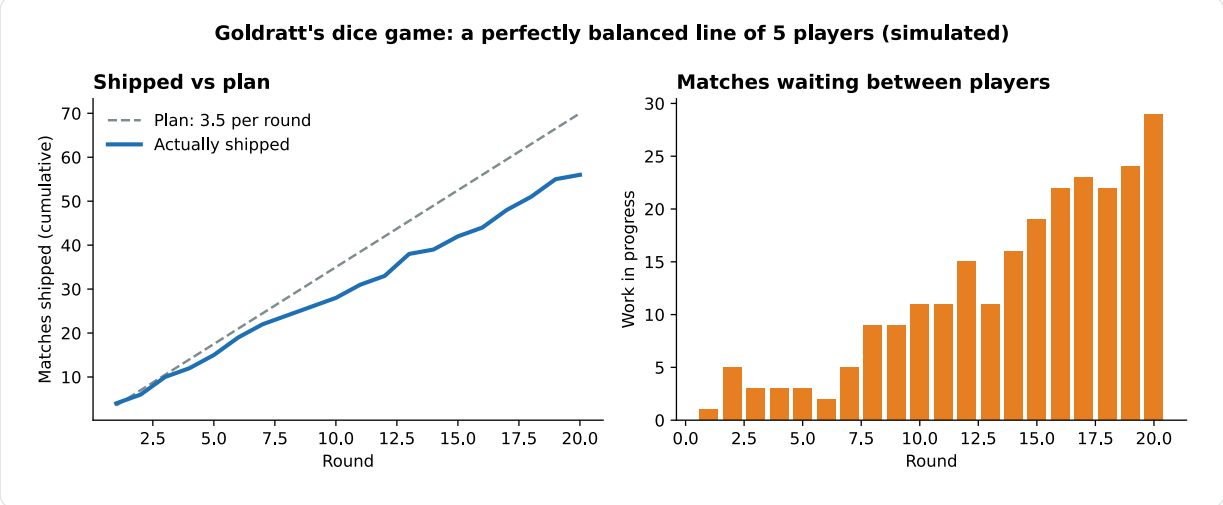


Figure 1 · Goldratt's dice game simulated: shipped versus plan, and growing work in progress

One simulated run of 20 rounds: the plan says 70 matches, the line ships 56, while work piles up between players. Different runs give different numbers; the pattern repeats.

Goldratt's conclusion surprises most managers: in a system with dependency and variation, **a perfectly balanced line, where everyone's capacity exactly matches demand, is a recipe for poor flow**. The closer to perfect balance, he writes, the closer to bankruptcy. Every real delivery system has dependency and variation. Chapters 4 and 5 show what that means for queues and utilisation.

Lead Time is a range, not a number

Ask "What's our lead time?" and most people expect one number. But the dice game already shows that the same work, done by the same people, takes different times.

Alexei Zheglov makes the point directly: **Lead Time is not a number, it's a probability distribution**, a range of outcomes with different likelihoods. And the shape is lopsided. Most items finish fairly quickly; a long tail of unlucky items takes much longer. Zheglov shows that for typical Lead Time shapes, the chance that an item takes longer than the most common value is **72–82%**. Team Atlas fits: its most common Lead Time was 3 days, yet about 75% of its items took longer.

The *Kanban Maturity Model* goes a step further. In some kinds of work the tail is so long ("fat") that **averages become meaningless**: a few extreme items dominate everything. Chapter 6 deals with this. For now, the practical rule is: **use averages to understand how the system behaves, and percentiles to make decisions and promises.**

Not all flows are the same shape

This is an idea most managers have never met, and it explains why so many improvement efforts fail.

Goldratt's school defines four types of flow, named after the letter they resemble:

TYPE	SHAPE	EXAMPLE	TYPICAL PROBLEM
I	One straight line	A request goes through analysis, development, testing, release	Work piles up in front of the slowest step
A	Many branches join into one	Many components assembled into one product or release	One missing component holds up the whole assembly
V	One input splits into many outputs	One platform turned into many variants or customer versions	One branch "steals" what another needed
T	A common base splits late into many variants	Standard parts combined into many customer configurations	Both stealing and missing components

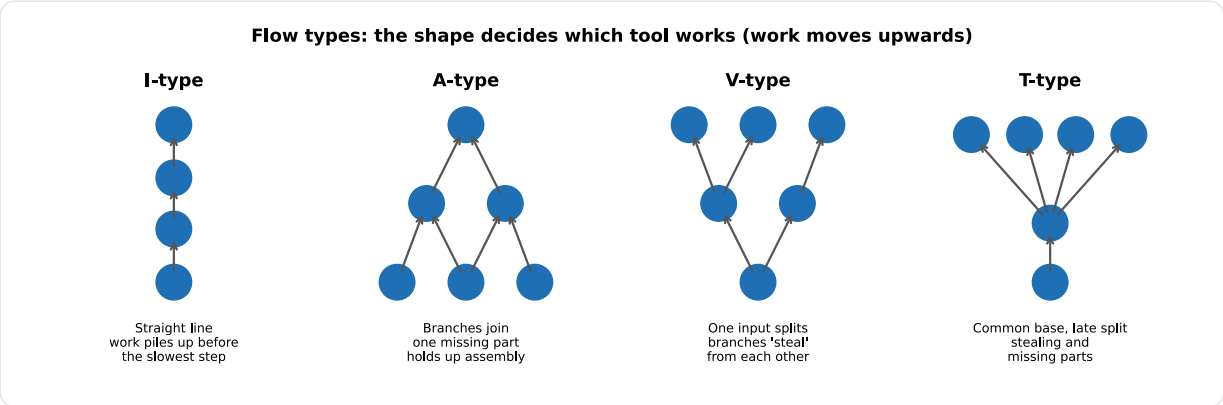


Figure 2 · The four flow types I, A, V and T

The key point is that **the type of flow decides which management tool works**. A tool designed for one shape fails on another, and when it fails, people are usually blamed.

The history follows the same pattern. Ford's system worked brilliantly for his A-shaped flow, because it was simple and stable: one car model, one colour, everything converging on final assembly. When General Motors arrived with a wide range of models, the flow started to branch, and Ford's advantage disappeared. Taiichi Ohno at Toyota then built a different system, with small batches and fast changeovers, to cope with more complex flows.

The longer the chain, the more fragile the flow. Chains of two steps are fairly stable; with three, problems start; with five or six steps, loosely managed, a project can become pointless before it is finished. Reduce the number of hand-offs in the chain, even if it costs more in direct effort. This is an observation from practice, not a law, but most managers will recognise it.

Where the clock starts

To measure flow, everyone must agree when the clock starts and stops. The *Kanban Maturity Model* and Zheglov define the start as the **commitment point**: the moment we agree to do the work and the customer can reasonably expect it. That is the **Committed** point in this guide. The clock stops when the item is **ready for delivery**.

Two related intervals are worth knowing:

- **System lead time**: from Committed to ready for delivery. This is what the team controls, and what this guide calls Lead Time.
- **Customer lead time**: from the customer's request to the moment they actually receive it. It includes time in the idea list before commitment and time waiting for release afterwards. It is what the customer experiences. Lead Time and WIP in this guide both stop at ready for delivery, so Little's Law holds between them. Work waiting for release is counted separately, as part of customer lead time.

Both matter. Measuring only system lead time can hide long waits before and after the team's part. Measuring only customer lead time blurs what the team itself can change.

WHAT THIS MEANS FOR YOU

- **Flow** is how customer value moves through the process: steadily, or in stops and starts. Good flow means little waiting.
- **Little's Law**: average Lead Time = average WIP ÷ average delivery rate. More work in the system means longer waits. *WIP and Lead Time are twins.*
- Most Lead Time is **waiting**. Typical flow efficiency is 5–15%.
- **Dependency plus variation** makes delays accumulate. A perfectly balanced, fully loaded system flows badly.
- Lead Time is a **range**, usually lopsided. Explain with averages; decide and promise with percentiles.
- **Flows have shapes** (I, A, V, T). Find out which shape yours has before choosing a tool.
- Fewer hand-offs make flow more stable.
- Agree where the **clock starts** (Committed) and stops (ready for delivery).

CHAPTER 2

Time is money: the cost of delay

The one number to get

If you take only one idea from this guide, Reinertsen says it should be this:

> *"If you only quantify one thing, quantify the cost of delay."*
>
> — REINERTSEN

Cost of delay is the money lost for each unit of time something is late, usually per week: "every week this is late costs us about €25,000."

Reinertsen calls it "*the golden key that unlocks many doors*". Without it, you can't say what a queue costs, whether spare capacity is worth paying for, whether smaller batches are worth the extra overhead, or which item should go first. Every one of those questions trades money for time.

> "*We simply have no business trading money for cycle time if we do not know the economic value of cycle time.*" (Reinertsen; his cycle time is our Lead Time)

Where the cost comes from

Delay costs money in different ways. The common sources:

- **Revenue that arrives later.** A service that would earn €40,000 a week earns nothing until it is live.
- **Savings that start later.** An automation that would save 200 support hours a month saves nothing while it waits.
- **Penalties.** A missed contract date triggers a penalty or credit.
- **A shorter selling window.** If a competitor, a season or a regulation sets the date, every week of delay eats into the time left to earn.
- **Risk that stays open.** A security fix that waits leaves the exposure in place.
- **Costs that keep running.** The team, the old system and the workaround all keep costing money until the work is done.

Not every delay costs the same: the shape of cost of delay

A single figure per week is a good start. The *Kanban Maturity Model* adds something important: cost of delay has a **shape** over time. It describes four typical shapes, and gives each one its own way of being handled (a **class of service**):

SHAPE	WHAT DELAY DOES	EXAMPLE	CLASS OF SERVICE
Expedite	Cost is immediate and rises steeply	A production outage; a security breach	Do it now, even ahead of other work
Fixed date	Little cost until a date, then a big jump	A regulatory deadline; a trade show; a seasonal launch	Start early enough to be sure of making the date
Standard	Cost builds up steadily over time	Most new features and improvements	Normal order, first in first out
Intangible	Little cost now; it may matter much later	Platform upgrades, technical debt, maintenance	Keep a share of capacity for it; it absorbs the peaks

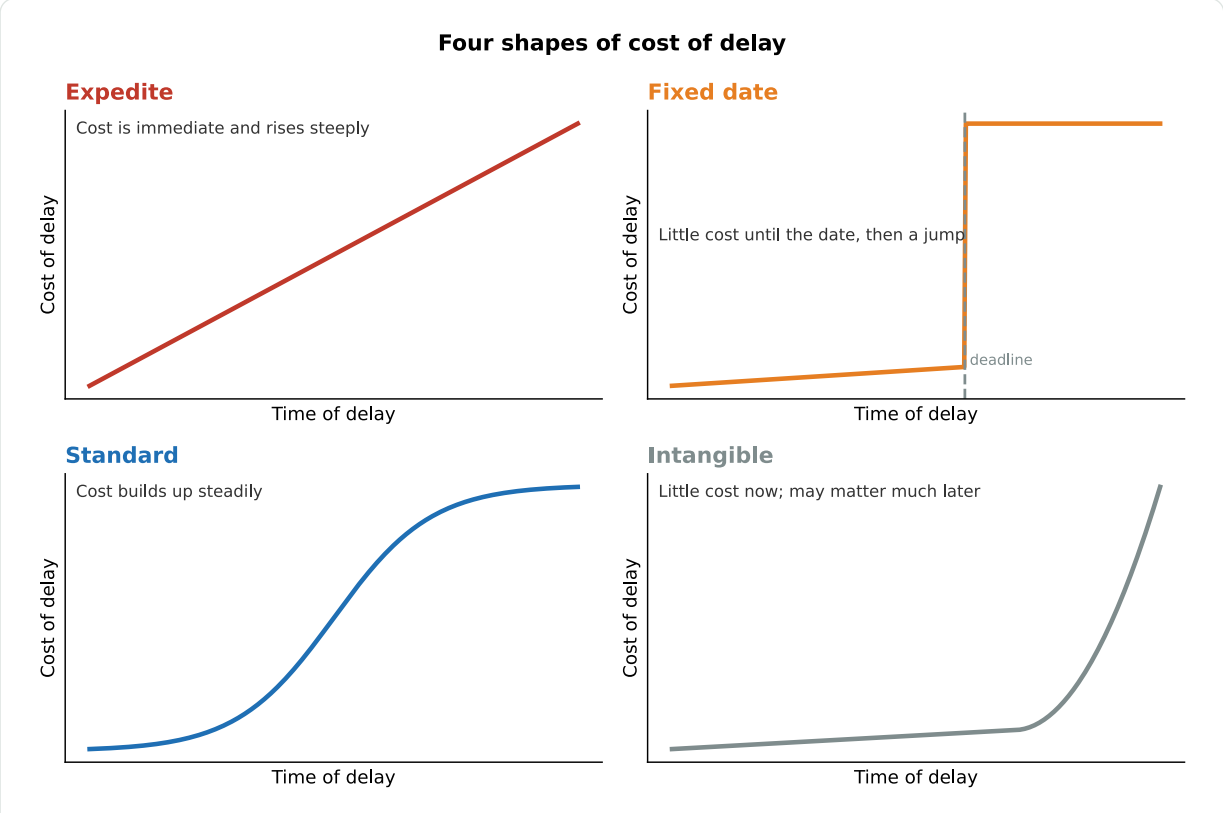


Figure 3 · Four shapes of cost of delay

Illustrative shapes, after the Kanban Maturity Model.

The shape tells you **when** delay starts to hurt, not just how much. A fixed-date item costs nothing to delay this week, and everything to delay past the deadline. An intangible item looks free to postpone, until it isn't.

Troy Magennis gives a simpler three-way split, which helps when deciding how much spare capacity a team needs (Chapter 5):

COST OF DELAY PATTERN	EXAMPLE	IMPLICATION
Almost zero (for us)	The customer has no alternative and will wait	High utilisation is economically acceptable
Temporary loss	Sales lost while a service is down; customers come back	Keep utilisation moderate
Permanent damage	A customer lost for good; a competitor launches first	Keep spare capacity; know in advance what you'll stop

David Anderson adds two shapes from software practice that managers will recognise. Some requirements **decay**: their value drains away the longer they wait, as the market moves on. Others have **no value before a date**: his example is tax legislation, where delivering early earns nothing, but delivering late is unacceptable.

Services, not a queue

Zheglov suggests a simple game. Ask colleagues to imagine they run a pizza shop and describe the variety of what they deliver. Most talk about sizes and toppings. But a pizza delivery chain actually offers **two different services**: pizza **as soon as possible**, where speed decides satisfaction, and pizza **at a set time**, where precision does: guests are coming at seven. No customer cares how many pizzas per hour the kitchen bakes. Each cares about their own order.

He describes a software group whose manager complained about "inconsistent velocity" and "too many interruptions". Mapped as services, the group was delivering four different things:

SERVICE	FOR WHOM	HOW REQUESTS ARRIVE	WHAT THE CUSTOMER EXPECTS
New product features	All customers	From a backlog	Predictable delivery
Production issues	All customers	Unpredictably	As soon as possible
Custom analysis reports	A few key customers	Random, somewhat predictable	Within 7 days
Feasibility checks for Sales	Colleagues in Sales	Unpredictably	Within 1–2 days

The "interruptions" were not interruptions. They were services with their own cost of delay. Once the manager saw that, the conversation moved from "protect the team" to "how much capacity does each service need?" That is the practical meaning of classes of service (Chapter 10).

Why we don't know it, and why that's fixable

The usual objection is that cost of delay is too hard to calculate precisely. Reinertsen's answer is one of his most useful principles:

> ***Even imperfect answers improve decision making.***
>
> — REINERTSEN

When people on a project work out the cost of delay together, even roughly, Reinertsen finds that the 50-to-1 spread of guesses typically narrows to about **2 to 1**. Still not precise, but good enough to tell a €500 decision from a €50,000 one.

Working it out also shows **which assumptions matter most**. Often one factor, such as a competitor's launch date or a contract penalty, drives almost everything.

A worked example: is four weeks worth €18,000?

Reinertsen gives an example most managers will recognise. A team could release a product before it is fully tested and let real use find the remaining defects.

- Finding the remaining defects in the lab would take **5 more weeks**. Real use finds them in **1 week**, so releasing early saves **4 weeks**.
- Fixing a defect after release costs ten times as much: **€20,000** instead of **€2,000**. With one such defect expected, releasing early adds **€18,000**.

Without cost of delay, this becomes an argument between quality and sales. With it, it's one line. (*Reinertsen poses the question in dollars; we use euros and add the cost of delay figures to show the arithmetic.*)

IF ONE WEEK OF DELAY COSTS...	FOUR WEEKS COST...	COMPARED WITH €18,000 EXTRA	DECISION
€2,000	€8,000	Less	Keep testing
€4,500	€18,000	Equal	Either way
€10,000	€40,000	More	Release early

THIS IS NOT AN ARGUMENT FOR ALWAYS GOING FASTER.

Sometimes the cost of going faster is higher than the cost of delay, and waiting is right. The point is that you can only know which by putting both in the same unit.

The cost you see and the cost you don't

David Anderson gives an example of how the visible cost of a delay understates the real one. A system test team of five testers loses its test environment for two days.

- **What cost accounting sees:** five testers × €100 an hour × 8 hours × 2 days = about **€8,000** of idle time.
- **What actually happened:** the test team was the bottleneck, delivering about 30 items a month. Two days out of a 20-day month means about **3 items lost for good**. They can't be made up, because the bottleneck was already full. Anderson takes an average cost of about **€10,000 per item** in this example (operating expense ÷ items delivered; his figure), so the loss is a minimum of about **€30,000**, before counting the delay to every item behind them.

(*Anderson uses dollars; we have converted for consistency.*)

The difference is not a rounding error. It's the difference between an idle-time cost and a **lost-throughput** cost: money the system will never earn. Chapter 3 explains why losing time at the bottleneck is so much more expensive than losing time anywhere else.

Cheaper on paper, worse in reality

Imagine two projects each earn **€2,000 a month** for **18 months** from today, after which the market changes and the profit disappears. One person-week of work costs €2,000.

- **Project 1** is done inside one department: two weeks of development, then two weeks of joint testing and fixing. Effort: 6 person-weeks (two of development, then two with developer and tester together), cost **€12,000**.
- **Project 2** needs four departments, one week each, strictly in sequence. Effort: 4 person-weeks, cost **€8,000**.

On paper, Project 2 is cheaper and just as fast: four calendar weeks either way. But in reality the picture is different: Project 1 actually finished in about **four months**, because its department already had a queue of "tasks promised long ago", "tasks set by the boss" and "just important tasks". Project 2, passing through four separate queues, took about **a year**.

	PROJECT 1	PROJECT 2
Cost	€12,000	€8,000
Actually finished after	about 4 months	about 12 months
Months of earnings left	14	6
Income	€28,000	€12,000
Profit	€16,000	€4,000
Return on investment	**133%**	**50%**

The cheaper project earns a quarter of the profit, and carries far more risk. Nothing in a cost-per-project report would show this. Only the cost of delay does.

Small improvements, large money

Troy Magennis gives an illustrative example of how much small improvements can matter when timing matters. A project missed a seasonal sales window and earned a cash benefit of only **$60,000**. Cutting its Lead Times by **10%** would have raised the benefit to about **$177,000**; cutting them by **20%**, to about **$265,000**.

A caution from the predictability school

Not everyone is comfortable with cost of delay as a precise tool. Daniel Vacanti argues that the value of a piece of work, and therefore its cost of delay, often **can't be known in advance** with any accuracy. He recommends using value and urgency when **choosing** what to commit to next, but not to keep reshuffling work once it is under way, because constant reprioritisation destroys predictability (Chapter 10).

The *Kanban Maturity Model* adds a further limit. Cost of delay works well for ordinary work, where the payoff is reasonably predictable. For bets whose payoff could be enormous or nothing, such as a new product that might become a hit, a single cost-of-delay figure is meaningless. The model's advice is to treat such work as a deliberate bet: give it a fixed share of capacity, rather than trying to rank it by an urgency number.

These cautions don't contradict Reinertsen. They tell you **where** cost of delay is most useful: in choosing and sequencing ordinary work, at the moment of commitment.

From one number to a rule anyone can use

Once you know the cost of delay, you can turn it into a **decision rule**: an exchange rate between time and money that people can use without asking permission.

Boeing 777. On the 777 airliner, weight was critical. Any designer could add up to **$300 to the cost of the aircraft to save one pound of weight**. Around 5,000 engineers could make that trade-off on the spot. Before rules like this, one subsystem might spend $5,000 to save a pound while another ignored savings at $50 a pound.

Buying time. Reinertsen describes organisations where each engineer may spend up to **$500 per week of schedule saved**, for up to four weeks, without approval. Managers and directors have higher limits.

> "*Control without participation is control without decision-making delays.*" (Reinertsen)

And when you need a decision from above, speak the right language. Reinertsen's principle: **to influence financial decisions, speak the language of money.** "We need to limit WIP because it improves flow" invites a debate. "Our WIP adds about three weeks to every release, and each week costs about €15,000" invites a decision.

TEAM ATLAS. Suppose Atlas's current release saves **€15,000 a week** once live (*an illustrative figure*). If starting two extra "quick" requests in parallel pushes the finish out by one week, that decision costs about €15,000. "Can you just squeeze this in?" now has a price.

How to start estimating cost of delay

1. **Pick one piece of work** that matters.
2. **Ask what it delivers when it's done**: revenue, savings, avoided penalties, reduced risk. Pick the one or two that dominate.
3. **Express it per week.**
4. **Identify its shape**: expedite, fixed date, standard or intangible. Is there a date after which value drops sharply?
5. **If revenue is hard to estimate, start with the team's running cost** (what the team costs per week, for example Atlas's €8,000). It is a lower estimate of the cost of delay: the team is paid whether the work is finished or not, so each week of delay costs at least this much. The real figure is higher, because it ignores lost revenue and risk.
6. **Agree the number with the people who make the trade-offs.**
7. **Use it in the next decision** that trades time for money, and revisit it when something important changes.

WHAT THIS MEANS FOR YOU

- **Cost of delay** is what a week of lateness costs. Get it, even roughly.
- It has a **shape**: expedite, fixed date, standard, intangible. The shape tells you when delay starts to hurt.
- A rough, **shared** estimate narrows disagreement from about 50:1 to 2:1.
- The **visible** cost of a delay (idle hours) is often far smaller than the **real** cost (lost throughput).
- A project that is **cheaper on paper** can earn far less if it waits in more queues.
- Use cost of delay mainly when **choosing** what to commit to next, not for constant reshuffling.
- Map your **services**, not just your queue. "Interruptions" are often services with their own cost of delay.
- Turn it into **decision rules**, and speak **money** when you need a decision.

CHAPTER 3

Throughput money: where the system makes money

Three measures instead of a hundred

In *The Goal*, Jonah tells Alex Rogo that he only needs three measures to run his plant. Not utilisation, not cost per part, not efficiency. Just three.

MEASURE	GOLDRATT'S DEFINITION
Throughput (T)	The rate at which the system generates money **through sales**
Inventory / Investment (I)	The money the system has invested in things it intends to sell
Operating expense (OE)	The money the system spends turning inventory into throughput

The goal, he says, is to **increase throughput while reducing investment and operating expense**, all at the same time. Profit and return follow:

```
Net profit = T − OE
Return on investment = (T − OE) ÷ I
```

This approach is called **throughput accounting**. It is not a replacement for financial reporting; it is a way of making **management decisions** that connects everyday choices to money.

The same idea for knowledge work

Goldratt developed this for factories. David Anderson, in *Agile Management for Software Engineering*, adapted it for software and IT work. His definitions are worth knowing, because they answer the question "what is inventory in work you can't see?":

MEASURE	ANDERSON'S VERSION FOR SOFTWARE
Throughput (T)	Revenue from what is delivered, minus costs that vary directly with each sale (licences, installation, commissions)
Investment (I)	Mainly the cost of **acquiring requirements**: the ideas, analysis and specifications the team has accepted. Ideas are inventory
Operating expense (OE)	Everything it costs to run the delivery system: salaries (including contractors), tools, facilities, financing

Three consequences follow, and each surprises most managers:

1. **Accepted ideas are inventory.** Every requirement the team has taken on but not delivered is money tied up, earning nothing. The more of them there are, the higher the investment, and the lower the return.
2. **Inventory goes stale.** Requirements that wait too long lose value. When they are abandoned, their cost is written off as a loss. Long queues of accepted work are not a harmless backlog; they are depreciating assets.
3. **Value counts only when delivered.** "Features built" is not throughput. Throughput happens when something reaches the customer and earns, or saves, money.

Anderson also gives an order of importance: **throughput first, then investment, then operating expense**. His reason: cutting cost has a floor (you can't go below zero), while increasing throughput has no natural limit. Steve Tendon, who built the TameFlow approach on Goldratt's ideas, makes the same point to managers: *reducing cost is limited, while increasing throughput is potentially unlimited.*

"Throughput" means two different things

Here the schools use the same word for different things, and it causes real confusion.

	THROUGHPUT (ITEMS)	THROUGHPUT (MONEY)
Used by	Vacanti, Magennis, the *Kanban Maturity Model*	Goldratt, Anderson, Tendon
Measures	How many items are delivered per week	How much money the system generates per week
Example	"Team Atlas delivers 4.4 items a week"	"Atlas's release generates about €15,000 a week in margin *(illustrative)*"
Good for	Forecasting, Little's Law, spotting instability	Business cases, investment decisions, priorities

Both are useful. They are not the same, and an increase in one doesn't guarantee an increase in the other: a team can deliver more items that earn nothing. **In this guide, "delivery rate" means items per week and "throughput" means money**, unless stated otherwise.

The hour that costs a fortune

The most practical consequence of throughput accounting concerns the **constraint**: the one step in the system that limits how much it can deliver. Chapter 7 covers constraints in detail. Here is the economic point.

In *The Goal*, Alex and his team find that the whole plant's output is limited by two resources: one particular machine and the heat-treatment furnace. Then they ask: what does an hour of the constraint cost?

Cost accounting says: the machine's running cost, a few dozen dollars an hour. Throughput accounting gives a very different answer. **An hour lost at the constraint is an hour lost for the whole system.** No other step can make it up. So the real cost of a constraint hour is the cost of running the **entire plant** for that hour. In the novel, the team calculates it as the plant's total operating expense divided by the constraint's available hours: about **$1.6 million ÷ 585 hours ≈ $2,700 an hour**.

That changes every decision about the constraint:

- **Idle time at the constraint** (a break, a missing part, a wait for approval) costs $2,700 an hour, not the operator's wage.
- **Defective work processed by the constraint** wastes constraint time that can never be recovered. Goldratt's advice: **inspect before the constraint**, not after.
- **Work at the constraint that nobody will buy** wastes the most expensive hours in the system.

The same logic applies to knowledge work. If one specialist team, one test environment or one approval board limits what the whole organisation delivers, an hour lost there is worth the whole organisation's hourly cost. Anderson's test environment example in Chapter 2 is exactly this: two days at the bottleneck cost not €8,000 of idle testers, but at least €30,000 of lost delivery.

And the reverse: **an hour saved at a step that isn't the constraint is, in Goldratt's words, "a mirage"**. It frees capacity at a step that already had spare capacity. The system delivers nothing more.

Making the case for investment

Throughput accounting also gives a simple test for spending money. Anderson's version: an investment that removes or eases the constraint is worth making if **return on investment after it is higher than before it**.

```
ROI before = (T − OE) ÷ I
ROI after  = (T + extra T − OE − extra OE) ÷ (I + extra I)
```

This immediately shows why many common investments don't pay:

- **Adding people at a step that isn't the constraint** only adds operating expense. Throughput doesn't move.
- **Hiring for the constraint** may pay many times over, even at a high salary, because every extra constraint hour is worth the whole system's hourly cost.
- **Automation at the constraint** (for example, test automation when testing is the bottleneck) can pay for itself quickly; the same automation somewhere else may not.

When cost accounting gives the wrong answer

Goldratt's novels are full of decisions where unit cost points one way and throughput money the other. Three types are worth recognising:

1. **Bigger batches look cheaper.** Running larger batches spreads setup time over more units, so the reported cost per unit falls. But it raises inventory and Lead Time, and the cash cost doesn't fall at all. *The Goal* shows a case where halving batch sizes made the reported unit cost rise, while the plant delivered more and faster (Chapter 8).
2. **Idle capacity looks like waste.** When a non-constraint step is idle, cost accounting sees inefficiency. Throughput accounting sees protective capacity, which keeps work flowing to the constraint (Chapter 5).
3. **Low-priced orders look unprofitable.** When there is spare capacity, an order priced below the "full cost" of a product can still add throughput, because the operating expense is being paid anyway. In *The Goal*, the plant wins a large contract this way. But the same story shows the limit: take on too many such orders without protecting the constraint, and overtime and late deliveries follow.

Use freed capacity, don't cut it

One last point, which matters for anyone leading change. When flow improves, capacity is freed. Lead Times fall, fewer people are needed to deliver the same amount, and some teams find themselves with spare time.

The instinct is to cut that capacity to save cost. Tendon warns against it: freed capacity is the reward for improving, and the best use of it is to **increase throughput**, take on more valuable work, or build the slack that keeps flow stable. Cutting it caps the gains at a one-off cost saving, and teaches people that improving flow costs them their jobs. Goldratt's *The Goal* makes the same point from the plant floor.

TEAM ATLAS. Suppose Atlas's release would generate a margin of €15,000 a week once live, and the team costs €8,000 a week to run. Atlas has about 40 items left in its release: roughly 6 in progress, the other 34 still in the backlog, not yet committed. In throughput terms: every week of delay costs about €15,000 of throughput; the 40 items are an investment that earns nothing until delivered; and any idle time at Atlas's slowest step is worth far more than its share of the €8,000. (*Illustrative figures.*)

WHAT THIS MEANS FOR YOU

- Run the system with three measures: **throughput (T)**, **investment (I)**, **operating expense (OE)**. Net profit = T − OE; return = (T − OE) ÷ I.
- In knowledge work, **accepted ideas are inventory**. They tie up money, and they go stale.
- **Throughput first.** Cost cutting has a floor; throughput doesn't.
- "Throughput" can mean **items** or **money**. Keep them apart.
- **An hour lost at the constraint is an hour lost for the whole system.** An hour saved elsewhere is a mirage.
- Judge investments by **return after versus return before**. Adding people away from the constraint only adds cost.
- Cost per unit can point the **wrong way**: big batches, idle capacity and low-priced orders.
- When flow frees capacity, **use it for throughput**, don't just cut it.

PART II

Why work waits

CHAPTER 4

Queues: the inventory nobody sees

The hare who lost by standing still

Reinertsen opens his chapter on queues with Aesop's tortoise and hare. The hare was much faster at running. It lost because it spent most of the race doing nothing.

Most delivery work is the hare. When someone is actually working on an item, it moves quickly. The trouble is the time when nobody is: it waits for a review, a test environment, a decision, an answer from another team. Reinertsen's diagnosis is that the real problem is "*periods of inactivity, not slow activities.*" Those periods are **queues**.

What a queue is

A queue is work that has arrived but isn't being worked on yet. It waits for a **resource**: a person, a team, an environment, a meeting, an approval. Queues are everywhere in knowledge work, even if nobody calls them that:

- items waiting for code review or testing,
- change requests waiting for the weekly approval board,
- tickets waiting for the one specialist who knows the old system,
- requests waiting for a decision from someone senior,
- finished work waiting for the next release window.

Reinertsen lists the places where he finds them most often. Testing is, in his words, "*probably the single most common critical-path queue in most R&D organizations.*" Specialists, management reviews and shared services follow close behind. The *Kanban Maturity Model* adds a pattern every large organisation will recognise: work **festers in the buffers between departments**, where nobody owns it.

Six kinds of damage

People usually think a queue only causes delay. Reinertsen lists **six** kinds of damage:

DAMAGE	WHAT HAPPENS
Longer Lead Time	Every item waits its turn, so everything arrives later
More risk	The longer work waits, the more the market, the requirements or the technology can change

DAMAGE	WHAT HAPPENS
More variability	Waiting times swing widely, so dates become harder to predict
More overhead	Waiting work needs status reports, follow-ups, re-prioritisation and meetings
Lower quality	Feedback arrives late, so mistakes are found late and copied into more work
Lower motivation	If the next step won't touch your work for four weeks, there is little reason to hurry

The quality point deserves an example. A programmer makes a wrong assumption about an interface. If someone checks the work the next day, one day of code is affected. If the check happens 30 days later, the mistake is built into 30 days of code. Reinertsen notes that this waste grows **faster than the delay itself**.

David Anderson adds the throughput-accounting view from Chapter 3: work waiting in a queue is **investment earning nothing**, and it goes stale. And Goldratt observes where queues form in a system with a constraint: work piles up in front of the constraint, because everything upstream can produce faster than it can absorb.

Swamping and starving

Anderson describes two opposite failures that queues create between steps:

- **Swamping**: a step receives far more work at once than it can handle. It becomes a temporary bottleneck. His example: an annual budget approval, where a whole year's work arrives on one day.
- **Starving**: a step has nothing to work on, because the step before it is holding work back or delivering it in large lumps.

Both are queue problems, and both have the same remedy: **smaller, more frequent hand-offs** (Chapter 8) and **limits on how much work enters** (Chapter 9).

Why nobody measures them

Chapter 0 explained why WIP is invisible. Queues are invisible for the same reason. Reinertsen's surveys found that practically every organisation measured how long work takes, and only about **2%** measured their queues. Organisations watch the outcome and ignore the cause.

Measuring queues with Little's Law

Chapter 1 introduced Little's Law: average Lead Time = average WIP ÷ average delivery rate. Reinertsen explains it with a water tank: 10 litres drained at 1 litre a minute takes 10 minutes; 20 litres takes 20 minutes. Nothing about the tap changed.

His examples: a development organisation with **50 projects** in progress that completes **10 a year** takes on average **5 years** per project. A test department with **50 tests** in progress that completes **10 a week** takes **5 weeks** per test.

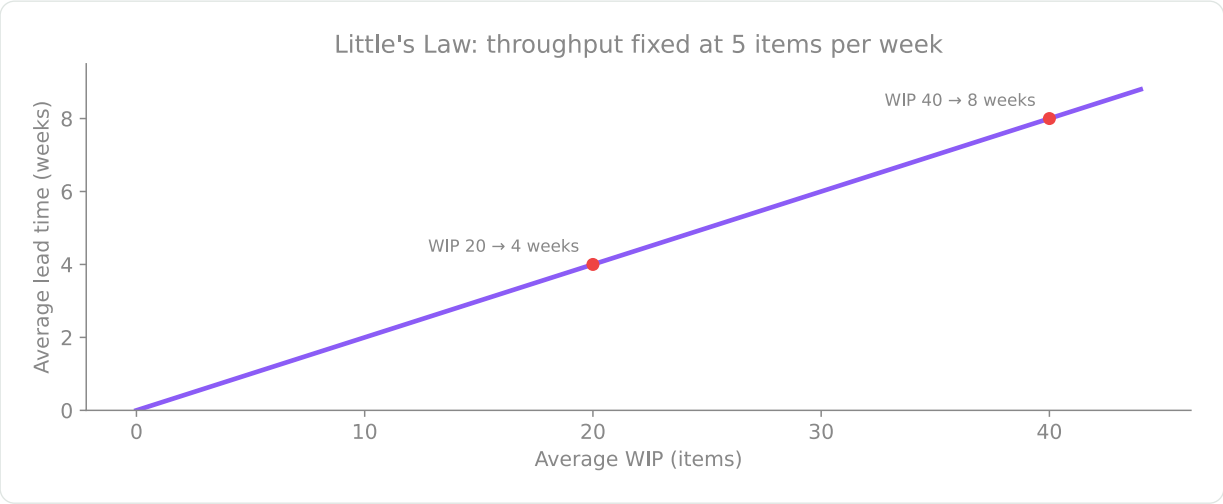


Figure 4 · Little's Law: with 5 items finished per week, more WIP means longer Lead Time

"We're working on everything"

Reinertsen describes what happens as soon as an organisation starts measuring queues. People quickly learn that queues look bad. Every item is suddenly "in progress"; nothing is "waiting". His answer is to stop trying to separate the two: measure **total WIP**, everything committed and not yet delivered. Little's Law works just as well on the total.

How much of the time is waiting?

Two simple ways to estimate it, both from Reinertsen:

1. **The "hot lot".** Give one item absolute priority everywhere, so it jumps every queue. Its Lead Time is almost pure working time. Compare it with the average Lead Time; the difference is queue time.
2. **Hours applied.** A process that applies **40 hours a week** and completes **10 jobs** gives each job **4 hours** of work. If a job takes **40 hours** of elapsed time to get through, it spends 36 hours waiting: **10% working, 90% waiting**.

These figures match what other authors report: Vacanti around 15% flow efficiency, Magennis 5–15% (Chapter 1).

The CFD: a picture of the queue

Reinertsen, Vacanti and the *Kanban Maturity Model* all recommend the **cumulative flow diagram** (CFD). It has two lines that only ever go up:

- the **top line** counts all items that have **arrived** (been committed) so far,
- the **bottom line** counts all items that have **departed** (been delivered) so far.

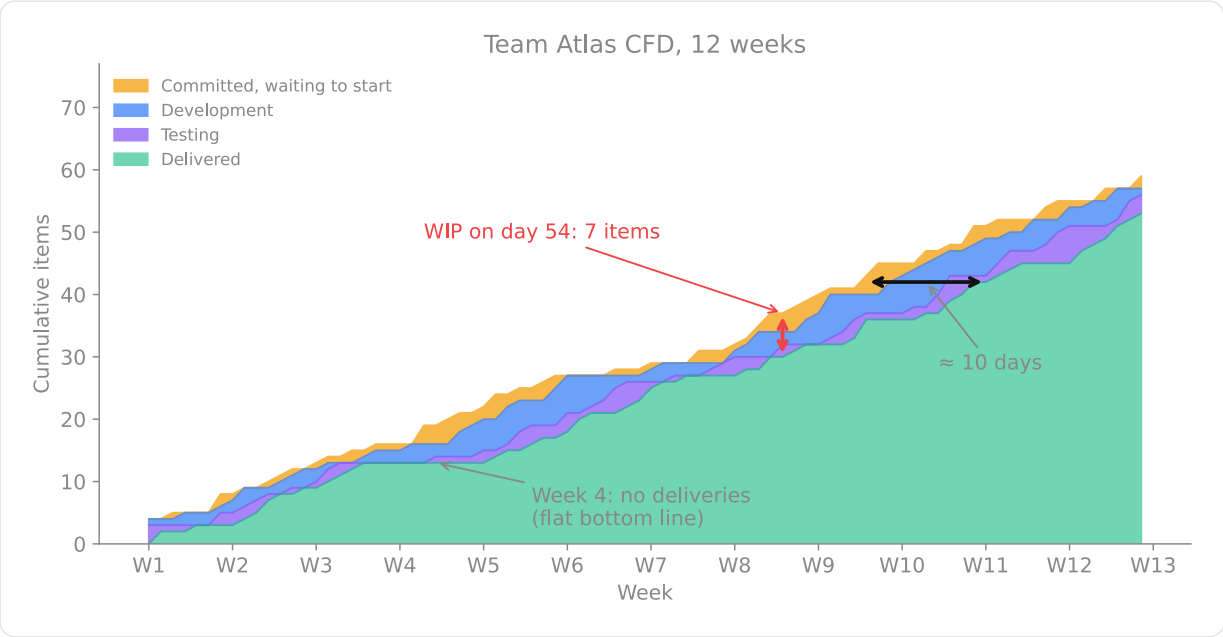


Figure 5 · CFD showing WIP as the vertical gap, Lead Time as the horizontal gap and delivery rate as the slope

Team Atlas, 12 weeks.

LOOK AT...	IT SHOWS...
The **vertical** gap between the lines	How much work is in the system that day
The **horizontal** gap	Roughly how long an item takes to get through
The **slope of the top line**	How fast work is arriving
The **slope of the bottom line**	How fast work is leaving

The CFD shows **why** a queue is growing: more work arriving, or less leaving. Vacanti adds two cautions. Keep the uncommitted backlog **off** the chart, because it distorts Lead Time. And use the CFD to **ask questions**, not to find culprits: a band that bulges shows where work collects, which is often in front of a step that is itself blocked further on.

Queues warn early; Lead Time warns late

> ***Don't control cycle time, control queue size.***
>
> — REINERTSEN; READ CYCLE TIME AS LEAD TIME

You only know an item's Lead Time when it is finished, after the delay has happened. Queue size you can see **today**. Reinertsen's example is an airport immigration hall: 100 people are waiting, and a large flight lands with **400 more**. The queue more than doubles instantly. Watching the queue reveals the problem **at once**; passengers' time through the hall shows it only much later.

Anderson agrees: WIP is a **leading** indicator, Lead Time a **lagging** one.

Queues don't fix themselves

Many managers assume a busy period will be followed by a quiet one. Reinertsen shows why that's wishful thinking, using a coin-toss experiment from the mathematician William Feller. Toss a coin 1,000 times; add one for each head, subtract one for each tail. The running total doesn't hover around zero. It wanders off and stays away for long stretches: there is a **50%** chance it never crosses back over zero in the last 500 tosses.

Queues behave like that total. Once one has drifted high, nothing pulls it back. Ten items above normal would need the equivalent of ten tails in a row to recover by luck: about **1 chance in 1,000**.

> *"We simply cannot rely on randomness to correct the problems that randomness creates."* (Reinertsen)

And a big queue multiplies every hiccup. With **20 jobs** in line, a 5-minute delay adds **100 minutes** of waiting across the queue; with **2 jobs**, only **10 minutes**. So: watch queues often, decide in advance how big you'll let them get, and act fast when they get there.

What a queue costs in money

Chapter 2 gave us the tool:

```
Cost of a queue = time waiting × cost of delay × number of items affected
```

Reinertsen's example: a queue on the critical path of **8 products** a year, each waiting **3 weeks**, with a cost of delay of **$100,000 a week**, costs **$2.4 million a year**. Nobody would sign off $2.4 million a year for a waiting room. Because the queue is invisible, organisations pay it without noticing.

WHAT THIS MEANS FOR YOU

- Most delivery time is **waiting**. The waiting happens in queues, often between departments.
- A queue does **six** kinds of damage, not one, and the work in it goes stale.
- Watch for **swamping** (too much at once) and **starving** (nothing to do).
- Measure **total WIP**, not just what people call "waiting".
- **Queues are a leading indicator; Lead Time is lagging.** Watch WIP and the CFD weekly.
- Queues **don't correct themselves**. Set limits in advance and act fast.
- **Price your queues**: waiting time × cost of delay × number of items.

CHAPTER 5

Utilisation and slack: why "everyone is busy" is expensive

The motorway at rush hour

At 6 a.m. the motorway is half empty. If one driver brakes, nobody notices. At 8 a.m. traffic is heavy but moving. One driver brakes a little too hard, the car behind brakes harder, and within a minute there's a jam that lasts an hour. The road didn't get narrower. It just got fuller. Past a certain point, small disturbances stop fading out and start piling up.

Teams, specialists, test environments and approval boards behave the same way.

What every school agrees on

This is one of the few topics where every author in this guide says the same thing, from different directions:

AUTHOR	THE POINT
Reinertsen	Queues grow sharply as utilisation approaches 100% (he says "exponentially"; strictly, the curve is hyperbolic)
Goldratt	*"A plant where everyone works all the time is extremely inefficient."* Non-constraint steps need spare (protective) capacity
Anderson	Idle capacity at steps that aren't the constraint is good, not waste
Magennis	The cost of waiting climbs steeply above about 80% utilisation, depending on the cost of delay
Vacanti	Slack is the only reliable way to stay predictable when work varies
Kanban Maturity Model	Optimise for flow efficiency, not resource efficiency; "relief from overburdening" is the first benefit managers should seek

Yet Reinertsen reports that executives in his courses, over 13 years, described their organisations as running at **98.5% utilisation** on average.

What happens to queues as utilisation rises

Queueing theory goes back to the Danish engineer Agner Erlang, who in 1909 worked out how many telephone lines Copenhagen needed when calls arrived at random. For work that arrives unpredictably and takes variable time, the answer is not a straight line.

> *Capacity utilisation increases queues exponentially.*
>
> — REINERTSEN

Reinertsen's rule of thumb: **every time you halve the spare capacity, you roughly double the queue.**

UTILISATION	SPARE CAPACITY	QUEUE, COMPARED WITH THE QUEUE AT 60%
60%	40%	1× (baseline)
80%	20%	about 2×
90%	10%	about 4×
95%	5%	about 8×

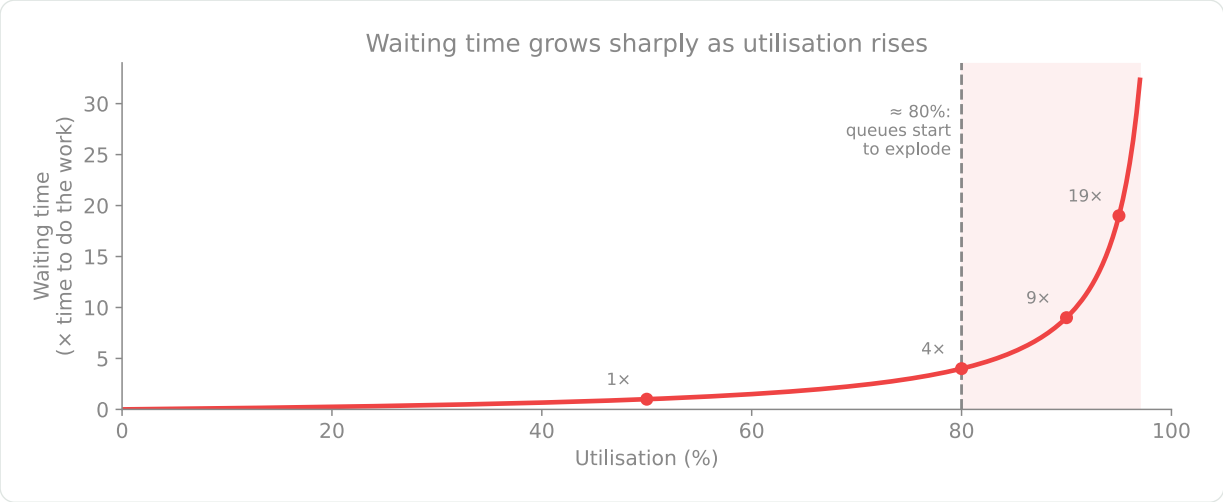


Figure 6 · Waiting time grows sharply above about 80% utilisation

Illustrative curve based on the standard queueing model.

The table above uses Reinertsen's rule of thumb, with the queue at 60% as the baseline; the exact model is steeper still. The next table uses a different baseline, the **pure working time** of an item, and is calculated from the standard model, so its numbers are larger. In the basic model, the share of an item's time spent waiting equals the utilisation. At 95% busy, about 95% of an item's time is waiting:

UTILISATION	LEAD TIME COMPARED WITH PURE WORKING TIME
50%	about 2×
75%	about 4×
90%	about 10×
95%	about 20×

Calculated from the standard model (Lead Time ÷ working time = 1 ÷ (1 − utilisation)). Real teams differ; the shape holds.

An example: the code reviewer. Picture one senior engineer who reviews all the team's code. A review takes about **one hour** of real work, and requests arrive at random through the day. Here is how long a developer waits, from asking for a review to getting it back, as the reviewer gets busier (*illustrative, calculated from the standard model above*):

REVIEWER IS BUSY	A ONE-HOUR REVIEW COMES BACK AFTER
50% of the time	about 2 hours: same morning
75% of the time	about 4 hours: same day
90% of the time	about 10 hours: tomorrow
95% of the time	about 20 hours: in two or three days

The review itself still takes an hour. Everything else is waiting. And the manager sees a reviewer who is "almost fully utilised", which looks like good news.

A team at 95% isn't 5% more productive than a team at 90%. Its items take **twice as long**. And the curve gets steeper near the top: the same extra request does about **25 times as much damage** at 95% as at 75%. Reinertsen calls the variability created by loading people this high "*a self-inflicted wound*".

Goldratt's version: protective capacity

Goldratt reaches the same conclusion through the dice game (Chapter 1). If every step has exactly the capacity needed, delays accumulate and early finishes are lost. In *The Goal*, Alex Rogo concludes that **steps that aren't the constraint need more capacity than the constraint**. Otherwise they can't catch up after a bad day, and the constraint starves.

He calls this **protective capacity**, and connects it to inventory: the more spare capacity upstream, the smaller the buffer of work needed in front of the constraint, and vice versa. You can buy stability with capacity or with inventory; you can't get it for free.

Anderson applies this to people. He plans for about **5.5 productive hours a day** and allows for around 15% of time lost to holidays, illness and interruptions, rather than assuming people deliver 8 or more hours a day all year. These are his own empirical figures, not research, but they show the principle: plans that assume full utilisation will be late.

Vacanti's FedEx example

Daniel Vacanti tells the story of FedEx, which keeps spare aircraft flying empty around its network every night. On paper they're pure waste. In practice, when a plane breaks down or weather closes an airport, a spare can step in, and packages still arrive overnight. FedEx's whole promise depends on that slack. Vacanti's point: **slack is what makes a promise reliable.** A team with no slack can't keep a delivery promise when anything goes wrong.

"80 is the new 100"

Many organisations have heard of companies that give engineers 20% of their time for their own projects, and try to copy it. Zheglov explains why it usually fails. The real purpose of "20% time" is to keep utilisation around **80%**, below the point where queues explode. His analogy: a computer whose processor is near 100% busy responds slowly to every click.

Mandating 20% time doesn't achieve that. As a practitioner put it to him: if 20% of capacity is reserved for other work, **80% becomes your new 100%**, and you are still in the utilisation trap. Nor does it help to justify slack through cost accounting, approval systems, timesheets, "innovation" or a fixed day each week. **You can't choose utilisation directly.** It results from demand and capability. Improve your capability to deliver and shape the demand, and slack appears. Its economic benefit is lower cost of delay, not more creativity. The practical levers are a **WIP limit** and **saying "no" or "later" at the commitment point** (Chapter 9); then run the work by watching queues, not utilisation.

When too full means less output

Beyond a certain point, overloading doesn't just slow things down: **output itself falls**. Reinertsen calls it **congestion collapse**. In October 1986, a fully loaded Internet link to the University of California, Berkeley, dropped its capacity by a factor of about **1,000**: delays caused resends, resends caused more load, more load caused more delays.

Organisations do the same in slow motion. When too much work is pushed in, people spend more time chasing, re-planning and reporting status, and late projects prompt the business to add more features or projects to make up lost revenue. The result is more busyness and less delivery.

How much spare capacity is right?

No single number fits every case. Magennis treats the common "keep utilisation below 80%" as a **guideline, not a law**. What matters is the cost of delay (Chapter 2):

COST OF DELAY PATTERN	SENSIBLE TARGET
Almost zero for us	High utilisation is acceptable
Temporary loss	Keep utilisation moderate, below about 80% most of the time

COST OF DELAY PATTERN	SENSIBLE TARGET
Permanent damage	Keep clear spare capacity; know in advance what you'll stop to make room

Reinertsen frames it as two costs that must balance: the cost of capacity, and the cost of delay caused by queues. The higher the cost of delay, the more spare capacity pays for itself.

Don't manage utilisation; manage the queue

Should managers set a utilisation target? Reinertsen says no, for a practical reason: **in knowledge work you can't measure utilisation accurately**. A 10% error in estimating the work plus a 10% error in estimating people's speed can leave you unsure whether a team is at **55% or 95%**: a difference of about **25 to 1** in queue size.

> *Don't control capacity utilisation, control queue size.*
>
> — REINERTSEN

His supermarket example: when more than **three** customers are queuing at a till, open another one; close it when the queue clears. Nobody calculates the cashiers' utilisation. Use utilisation for **long-term capacity planning**; use WIP and queues for running the work.

TEAM ATLAS. Atlas committed to 4, 4, 4, 5, 6, 1, 3, 8, 6, 6, 4, 4 items a week: 55 started against 53 delivered. In 7 of its 12 weeks it started more work than it finished, and its weekly delivery swung from 0 to 8. Nobody measures Atlas's utilisation, and nobody needs to: its WIP and CFD already show a loaded team.

WHAT THIS MEANS FOR YOU

- **Full utilisation is expensive.** All the authors agree.
- **Halve the spare capacity, double the queue.** At 95% busy, items spend most of their time waiting.
- Steps that aren't the constraint need **protective capacity**, or the constraint starves.
- **Slack is what makes promises reliable** (FedEx).
- Overload can **reduce output**, not just slow it down.
- The right amount of slack depends on the **cost of delay**.
- **Mandating** 20% free time doesn't create slack; 80% becomes the new 100%. Improve capability and shape demand.
- Manage the **queue (WIP)**, not utilisation.

CHAPTER 6

Variability, uncertainty and fat tails

Three ways to spend €15,000

Reinertsen offers a choice between three experiments, each costing €15,000. (*He uses dollars; the numbers work the same way.*)

OPTION	CHANCE OF SUCCESS	PAYOFF IF IT WORKS	EXPECTED VALUE AFTER THE COST
A	50%	€100,000	**€35,000**
B	90%	€20,000	€3,000
C	100%	€16,000	€1,000

Option C is completely predictable and the worst choice. Option A fails half the time and is worth more than ten times as much as B. Most organisations would pick C.

This chapter is about variability: where the schools agree it's harmful, where some argue it's valuable, and how to read it in your data.

Not all variability is bad

Many quality programmes aim to remove variability everywhere. Reinertsen reports that **65%** of developers he surveyed thought it desirable to eliminate as much of it as possible. In a factory making the same part 10,000 times, that makes sense. In knowledge work, whose job is to create something **new**, it doesn't:

> "*We cannot add value without adding variability, but we can add variability without adding value.*" (Reinertsen)

What matters is the **payoff**. When possible gains are much larger than possible losses, more uncertainty can be worth more. But Reinertsen is equally clear that variability should be **neither minimised nor maximised**: "*Never confuse seeking economic payoffs with seeking variability.*"

Troy Magennis adds a warning from practice: "*Driving for standardization in creative pursuits often leads to erosion of quality and inventiveness as teams strive to meet the pre-set control limits at great effort and at any cost.*" For anyone working on quality with managers, this is the key distinction: **variability is not the same as poor quality**.

Reduce the consequences first

Where variability does hurt, Reinertsen's advice surprises people from a quality background:

> ***Reducing consequences is usually the best way to reduce the cost of variability.***
>
> — REINERTSEN

Toyota's looms stop automatically when a thread breaks (**jidoka**). That doesn't make threads break less often; it stops one broken thread from ruining a whole roll. In knowledge work, the equivalent is **fast feedback**: a wrong assumption caught after one day instead of thirty still happened, but did far less damage.

Goldratt adds a placement rule: **check quality before the constraint**, not after. A defect caught before the constraint costs a little rework; a defect processed by the constraint wastes its most expensive hours (Chapter 3).

Why padded estimates still produce late projects

Goldratt's *Critical Chain* asks a question every project manager has wondered about. If people pad their estimates, why are projects still late?

He starts with how people estimate. Task times are lopsided, like a commute: usually 30 minutes, sometimes 90. Asked for an estimate, people don't give the most likely time; they give one they're 80–90% sure of. For a lopsided distribution that can mean hidden safety of **200% or more** of the likely time in each task. So where does all that safety go? Goldratt identifies three leaks:

1. **Student syndrome.** If there's safety in the estimate, people start late. Like students who start revising the night before the exam, they use the safety up front, then hit a problem with no time left.
2. **Parkinson's law.** Work expands to fill the time available. A task estimated at 10 days rarely finishes in 6, even if it could: there's always more polishing, and finishing early can look like the estimate was padded.
3. **Delays pass on; early finishes don't.** When one step finishes late, the next starts late. When a step finishes early, the next person usually isn't ready to start. This is the dice game again.

He adds a fourth: **multitasking**. Suppose someone has three tasks, each of 10 days. Done one after the other, the first finishes on day 10, the second on day 20, the third on day 30. Switched every 5 days, they finish on days 20, 25 and 30. Counted from the day work on each task began, **each now takes 20 days instead of 10**. The customer sees a smaller but still real effect: the average finish date moves from day 20 to day 25, **25% later**, and the first task is twice as late. All this before counting the time lost to switching itself.

Goldratt's remedy is to remove the safety from individual tasks and put a **shared buffer** at the end of the project (Chapter 11). Reinertsen agrees with the diagnosis: *"The more granular the schedule, the larger the schedule reserves."*

Buffers: two views

Here the schools seem to disagree, but they're closer than they look.

	REINERTSEN	GOLDRATT
Warning	*"A buffer converts uncertain earliness to certain lateness."* In his example, a 90% date is three months later than a 50% date, and that lateness is certain	Safety hidden inside every task gets wasted by student syndrome, Parkinson's law and multitasking
Remedy	Don't trade time for predictability unless the cost of delay justifies it	Remove hidden safety from tasks; pool it into one visible buffer for the whole project

Both reject **hidden padding in every task**. Goldratt keeps a buffer, but makes it one, shared and visible; Reinertsen asks what it costs. Chapter 11 shows how to combine them.

Reading variability in your data: noise or signal?

Daniel Vacanti's second volume of *Actionable Agile Metrics* adds a practical question managers face every week: when a number goes up or down, **does it mean anything?**

He borrows from Walter Shewhart and Donald Wheeler. Every process has **routine variation**: ordinary ups and downs from many small causes. Occasionally there is **exceptional variation**: a real change with a specific cause, such as an outage, a key person leaving or a new type of work. Managers make two kinds of mistake:

MISTAKE	WHAT IT LOOKS LIKE	WHAT IT COSTS
1. Reacting to noise	"Lead time went up this week; why?" when it's routine variation	Wasted investigations, blame, tampering that makes the process less stable
2. Ignoring a signal	Treating a real change as "just a bad week"	Problems grow until they become crises

Vacanti recommends simple **process behaviour charts** that draw limits around routine variation, so that only points outside them call for investigation. His rule for managers: before asking "*who is to blame*?", ask "*is this a signal*?" And one more: forecasts made from an **unstable** process, one with exceptional variation, can't be trusted. Fix the causes of instability first, then forecast.

Lopsided and fat-tailed Lead Times

Chapter 1 showed that Lead Times are lopsided: a hump of quick items and a long tail of slow ones. Magennis explains why. Delivery time is the work itself **plus a series of delays that may or may not happen**: waiting for a specialist, an environment, another team. Most items hit a few delays, some none, some many. The result is a skewed shape, not a bell curve, and the familiar "average ± 3 sigma" limits are simply wrong for it. (*Sigma*, the standard deviation, measures spread around the average; "± 3 sigma" is meant to cover almost all outcomes, but only for bell-shaped data.)

The *Kanban Maturity Model* takes this further. It distinguishes two worlds, using terms from Nassim Taleb:

	THIN TAILS ("MEDIOCRISTAN")	FAT TAILS ("EXTREMISTAN")
Shape	Lopsided, but extreme values are rare and bounded	A few extreme values dominate everything
Averages	Usable to describe behaviour	Meaningless; one outlier can move them anywhere
Typical causes	Ordinary waiting and rework	Many external dependencies, frequent expediting, many defects
What to do	Use percentiles for promises	Shorten the tail first: remove dependencies, limit expediting; use high percentiles

The model makes a strong claim: in a fat-tailed world, any method that relies on an average, including the standard **ROI = value ÷ cost** used to rank projects, becomes invalid. Both Vacanti and the *Kanban Maturity Model* note what fattens the tail: **external dependencies**, **expediting** and **defects**. These are, at least partly, management choices.

What the shape says about the service

Zheglov, who studied many real Lead Time data sets, reads the shape of the distribution as a description of the service's risks. In plain terms he finds three domains:

DOMAIN	SHAPE	TYPICAL CAUSES	WHAT IT MEANS
Fat-tailed	A long tail; items that have waited long tend to wait even longer	Poor prioritisation, hidden classes of service, unmanaged dependencies on unpredictable services; common in IT operations and support	Unpredictable; likely not fit for the customer's purpose. Trim the tail first
Well-managed knowledge work	Lopsided, moderate tail	Customer feedback built into the work; delays and dependencies managed	The healthy middle; most product development
Very thin-tailed	Little variation at all	Routine, repeatable work; phase-gated processes (often with long averages)	Surprisingly, a warning: little real knowledge work is happening, and the service may be ripe for disruption

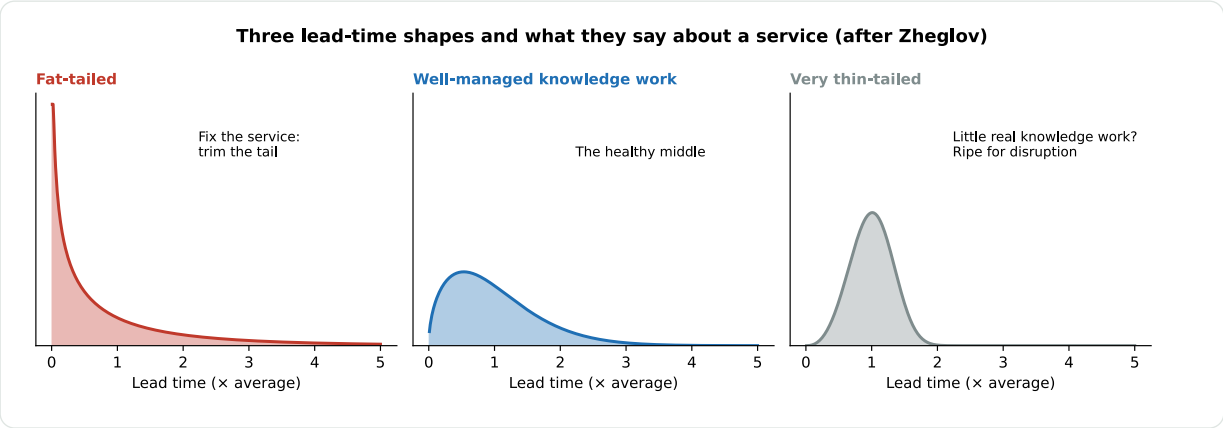


Figure 7 · Three Lead Time shapes: fat-tailed, well-managed, very thin-tailed

Illustrative shapes, scaled so the average is 1. They are drawn as Weibull curves, a standard family of lopsided distributions often used for Lead Times; a lower "shape" value (0.7, 1.5, 3.5) means a fatter tail.

In the data sets he observed, the tail in support and operations could be extreme: the 99th percentile around **15 times** the median. He adds two cautions. The most common Lead Time can be only **20–40%** of the average, so the value people remember is far too optimistic. (Atlas's most common Lead Time is 3 days; its median is 9.) And when you're unsure which shape you have, **assume the fatter tail**: overestimating how thin the tail is understates the risk a lot, while the opposite mistake overstates it only a little.

TEAM ATLAS. Atlas's Lead Times are lopsided with a moderate tail, Zheglov's healthy middle: median 9 days, 85th percentile 16, 95th percentile 23. Its longest items were mostly delayed by dependencies on other teams, which is where its manager should look first.

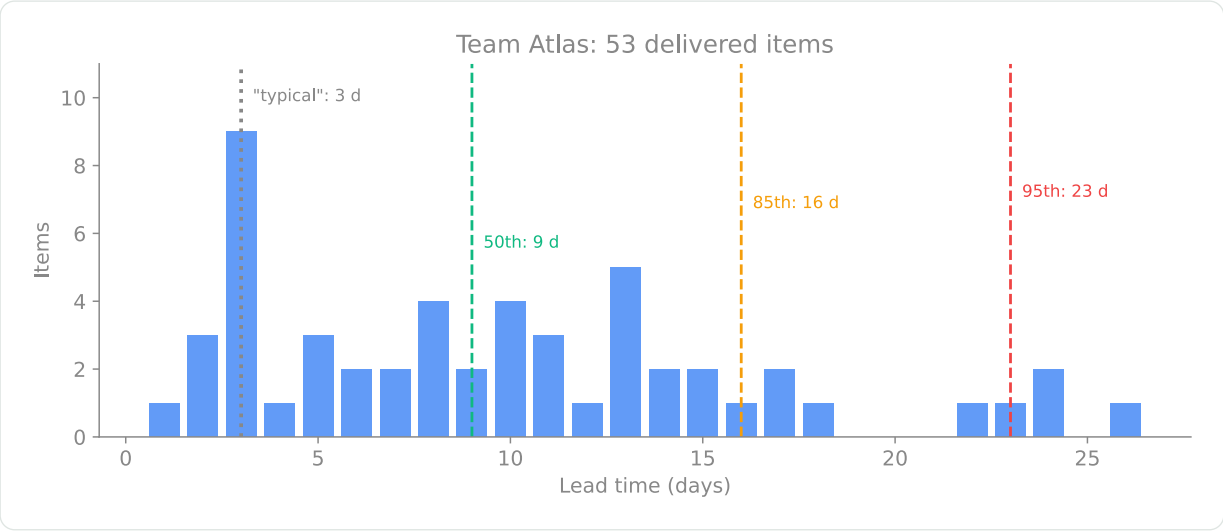


Figure 8 · Team Atlas Lead Times: median 9, 85th 16, 95th 23 days

Team Atlas, 53 delivered items. Most finish within about two weeks; a thin tail stretches past three weeks.

Plans, conformance and new information

The last belief linked to variability is **worship of conformance**: the plan is right, and any deviation must be corrected. Reinertsen's example: a feature planned as **1 week** of work valued by **50%** of customers turns out to need **10 weeks** and interest only **5%**. Its cost-to-benefit ratio has changed by a factor of **100**.

> *"To blindly conform to the original plan when it no longer represents the best economic choice is the act of a fool."* (Reinertsen)

Vacanti adds a related warning about targets. When management sets arbitrary targets ("lead time must be under 10 days"), teams find ways to hit the number, often by distorting the data, without improving the system. As Goldratt put it, measures shape behaviour (Chapter 13).

For managers, the question in a review changes from "Are we on plan?" to **"Given what we know now, is the plan still the best economic choice?"**

WHAT THIS MEANS FOR YOU

- **Variability is not the same as poor quality.** Judge it by its payoff.
- It's usually cheaper to **reduce consequences** (fast feedback, checking before the constraint) than to reduce variability.
- **Padding in every task gets wasted** by student syndrome, Parkinson's law, passed-on delays and multitasking. Multitasking can double the time each task takes once started, and pushes average delivery later.
- Reinertsen and Goldratt agree: **no hidden safety in each task**. Make any buffer visible and know what it costs.
- Before reacting to a number, ask: **noise or signal?**
- Lead Times are **lopsided**; some are **fat-tailed**. Dependencies, expediting and defects fatten the tail.
- When unsure of the shape, **assume the fatter tail**. A very thin tail can itself be a warning.
- Ask: **is the plan still the best economic choice?**

PART III

Levers

CHAPTER 7

Constraints: where to focus

Herbie

In *The Goal*, Alex Rogo takes his son's scout troop on a hike. The column keeps spreading out. The fast boys at the front race ahead; the gaps grow; the troop arrives late. The slowest boy, Herbie, carries a heavy rucksack.

Alex tries two things. First he puts Herbie **at the front**, so nobody can get ahead of him: the column stops spreading. Then he **lightens Herbie's load**, sharing it among the others: the whole troop speeds up.

That is Goldratt's Theory of Constraints in one story. Every system has a **constraint**, the step that limits how much the whole can deliver. Speeding up anything else makes no difference. Speeding up the constraint speeds up everything.

The five focusing steps

Goldratt turns the story into five steps:

STEP	WHAT IT MEANS	KNOWLEDGE WORK EXAMPLE
1. Identify the constraint	Find the step that limits the whole system	The one test environment; the architecture review board; a single specialist
2. Exploit it	Get the most out of it as it is: never let it idle, never waste it on bad or unnecessary work	Check quality *before* testing; keep a small queue ready so testers never wait; remove meetings from the specialist's day
3. Subordinate everything else	Make every other step serve the constraint, even if that means they're sometimes idle	Developers don't start more than testing can absorb; when testing is overloaded, they help test
4. Elevate it	Only now, invest to increase its capacity	Automate tests; hire or train a second specialist
5. Repeat	Once it's no longer the constraint, find the new one. *Don't let inertia become the constraint*	The bottleneck moves to release management; start again

The order matters. Steps 2 and 3 cost almost nothing. Step 4 costs money. Most organisations jump straight to step 4 ("we need more testers") without trying steps 2 and 3.

Drum, buffer, rope

Goldratt's way of subordinating the rest of the system to the constraint is called **drum–buffer–rope**:

- **Drum**: the constraint sets the pace for everyone, like a drummer in a marching column.
- **Buffer**: a small, deliberate amount of work kept in front of the constraint, so it never runs out. In *The Goal* the plant keeps about **three days** of work in front of each bottleneck and checks it daily. A "hole" in the buffer shows a problem upstream before it reaches the constraint.
- **Rope**: work is released into the system only at the rate the constraint can absorb, like tying the head of the column to Herbie's pace with a rope.

In *Critical Chain*, Goldratt sums up the consequence: **WIP and Lead Time are twins**. Tie the release of work to the constraint, and WIP stops growing, so Lead Time stops growing too. This is a WIP limit with a reason behind it (Chapter 9).

Constraints in knowledge work

David Anderson applied this to software teams in 2003. His examples of the constraint include a developer team, a test team and even access to a shared code repository. Two of his points matter for managers:

- **Rank work by money per hour of the constraint.** If testing is the constraint, the most valuable work is the work that earns the most per hour of *testing*, not per hour of development.
- **Adding people is not always elevation.** Adding developers to a late project often makes it later, because new people consume the time of experienced ones (Brooks's law). Anderson treats this as a failed attempt to elevate a constraint that wasn't where people thought it was.

The *Kanban Maturity Model* adds two practical refinements:

- **Place the constraint after the commitment point.** If the constraint is upstream, before work is committed, people at the downstream steps fill their time with upstream work, and the real slack in the system becomes invisible.
- **Route low-risk work around the constraint.** Not every item needs to pass through the bottleneck: some low-risk changes can take a lighter path, freeing constraint time for work that needs it.

Daniel Vacanti, generally sceptical of applying the Theory of Constraints to knowledge work, tells a story where it worked. At Siemens, the CFD showed stories piling up in front of **testing**. Management invested in **test automation**, and a forecasting model recommended hiring **one tester instead of two programmers**. That is "identify, then elevate", driven by data.

Zheglov describes a case closer to most IT organisations. A performance-testing team was so overloaded with requests that about **three years** of work had queued up. Before anyone hired more testers, the five focusing steps suggested cheaper moves: let low-risk features skip full performance testing, have the experts **coach** product teams instead of doing every test themselves, and replace generic "please test this" requests with specific questions the teams wanted answered. Only after that would elevating (hiring) make sense.

How to find the constraint

In a factory, you can see the pile of parts. In knowledge work, Steve Tendon's TameFlow approach suggests three places to look:

WHERE TO LOOK	QUESTION	WHAT IT SHOWS
Past	Which state has the longest average time?	Where work has been waiting
Future	Which queue holds the most time-worth of work?	Where work will wait
Present	Which team or step is most behind right now?	Where to act today

A simpler check: on a CFD, the constraint is usually the step just after the band that keeps getting fatter.

Where the schools disagree

Reinertsen is sceptical of starting with the bottleneck:

> ***Reduce batch size before you attack bottlenecks.***
>
> — REINERTSEN

His reasons: in knowledge work, bottlenecks **move**: this week testing, next week one architect, the week after the approval board. And Toyota's famous gains, he argues, came mainly from smaller batches, not from adding bottleneck capacity. Tendon calls the same problem the "**moving constraints syndrome**". *Where* the constraint sits also changes which tools work at all (Chapter 1).

SITUATION	BETTER FIRST LEVER
One **stable, visible** constraint (a test lab, a scarce specialist, a regulatory review)	**Goldratt**: identify, exploit, subordinate, buffer
Bottlenecks that **move** from week to week	**Reinertsen**: smaller batches and local WIP limits first (Chapters 8–9)
Constraint is a **management policy** (approval rules, annual budgets, matrix reporting)	Change the policy (Chapter 14)

That last row matters most. At the end of *The Goal*, the team finds that the root constraint was not a machine at all, but **management policy**. Goldratt calls these **policy constraints**. They are often the cheapest to remove and the hardest to see.

WHAT THIS MEANS FOR YOU

- Every system has a **constraint**. Improving anything else doesn't improve the whole.
- Follow the order: **identify, exploit, subordinate, elevate, repeat**. Steps 2 and 3 are almost free.
- **Drum–buffer–rope**: release work at the constraint's pace and keep a small buffer in front of it.
- Check quality **before** the constraint. Rank work by **money per constraint hour**.
- If bottlenecks **move**, start with batch size and WIP limits instead.
- Many constraints are **policies**, not resources.

CHAPTER 8

Batch size: the cheapest lever

Two hundred drawings

Reinertsen tells a story about engineering drawings. All **200** were reviewed together, in one day, because manufacturing didn't want to see anything until the design was finished. But they had been completed over **10 weeks**. The first 20 waited **9 weeks** for review; a wrong assumption in one of them was copied into the next 180.

The fix: review every week. "*Every Wednesday afternoon at 1:00 PM, we review all the drawings completed in the last week.*" No invitations, no scheduling, almost no overhead.

A **batch** is how much work is bundled before it's passed on, checked, released or decided. Every school in this guide agrees that most organisations' batches are too large.

Where batches hide

- a release with 40 features,
- annual funding for a whole programme,
- a phase gate that hands over **100%** of the requirements before design starts,
- a test cycle once at the end,
- a monthly approval board,
- a lessons-learned review only when a project closes.

Reinertsen: "*We cannot make batch size any higher than 100 percent.*" David Anderson adds a money view: a release every **30 days** caps the organisation's exposure at 30 days of operating expense invested in work not yet earning. A six-month release puts six months at risk.

What smaller batches do

SMALLER BATCHES...	WHY
shorten Lead Time	Less work waits for the rest of its batch; by Little's Law everything flows through faster
smooth the flow	A busload of 80 tourists swamps a restaurant; the same people in small groups don't
speed up feedback	Mistakes are found after days, not months
reduce risk	Less time exposed to change, and less lost when something fails

SMALLER BATCHES...	WHY
cut overhead	Fewer items to track and report on at once
lift motivation	One module due Friday feels urgent; one of 100 due in 100 days doesn't
prevent disproportionate slippage	In one organisation Reinertsen describes, projects planned to take twice as long slipped about 16 times as much in absolute terms
stop the death spiral	Big projects attract more scope and become impossible to stop

Reinertsen stresses that smaller batches cut Lead Time **without hiring anyone or turning work away**, often by an order of magnitude.

Goldratt's evidence: the accounting paradox

In *The Goal*, the plant halves batch sizes at the steps that aren't the constraint. WIP and Lead Time halve. But the reported **cost per part goes up**, because each setup is now spread over fewer parts. In the novel's numbers: a batch of 100 with a 2-hour setup and 5 minutes per part gives 6.2 minutes per part; a batch of 50 gives 7.4. Cost accounting says the plant got worse. In cash, nothing got more expensive, and the plant delivers more, faster. This is exactly the proxy trap from Chapter 0.

Goldratt's *It's Not Luck* adds two business cases where batch size changes the economics of a whole company:

- **Smaller, more frequent deliveries to customers.** A printing business offers clients delivery every two weeks instead of large six-month orders, with free cancellation. Clients hold far less stock and stop paying for printed material they never use.
- **Daily replenishment instead of large, infrequent orders.** Stores that used to order about every four months move to daily replenishment. In the novel, the time customers take to pay falls from about **116 to 45 days**, releasing tens of millions in cash, even though sales dip at first.

(These are cases from a business novel. Use them as illustrations, not benchmarks.)

There is a right size, and it's smaller than you think

Every batch has a **transaction cost**: the fixed effort to release, test, review or approve, however much is in it. Small batches pay it more often. Large batches pay it rarely, but items wait longer: a **holding cost**. Magennis and Reinertsen both describe the total as a **U-shaped curve**.



Figure 9 · Setup cost, holding cost and total cost by batch size

Illustrative.

Three properties make it manager-friendly:

1. **It's forgiving.** The bottom is flat: missing the optimum by 10% costs less than 1% extra.
2. **It's reversible.** You can step back.
3. **You can creep up on it.** Release a little more often, check, repeat.

Reinertsen warns that **the calculated optimum is almost always too large**, because large batches carry hidden costs the calculation ignores. His rule of thumb: go about **30% smaller** than the calculation suggests. The risk is small: even landing a third below the true optimum raises total cost by only about **8%**.

Make small batches affordable

> ***Reducing transaction cost per batch lowers overall costs.***
>
> — REINERTSEN

Changing the heavy dies in a car-body press used to take **24 hours**, so factories ran huge batches between changes. Japanese engineers cut it to **under 10 minutes**. Once changeovers were cheap, small batches became economical.

In knowledge work, the equivalents are automated testing and deployment, standard environments, a fixed rhythm of reviews (Chapter 12), and systems loosely coupled enough that one part can change without the rest. The *Kanban Maturity Model* puts it as a rule: **the cost of switching sets how often you can afford to deliver.** Lower it, and you can deliver more often.

The hand-off matters most

Reinertsen distinguishes how much work is **done** in one go from how much is **handed on** at once. A tester can run 1,000 tests in one session and still pass each result to the developer as soon as it's ready.

> *The most important batch is the transport batch.*
>
> — REINERTSEN

This is often the easiest place to start: people don't change how they work, only how often they share it.

A milder view

Daniel Vacanti is less absolute. Batching, he says, is **not inherently bad**; managers should look at its effect on Lead Time and predictability before deciding. He also gives a quality reason to prefer small batches: in the Siemens case, large annual releases meant defects were discovered late, when they were most expensive to fix. The schools differ in emphasis, not in direction.

How small is small enough?

Teams working in sprints often find that stories started in a sprint aren't finished in it. Zheglov's diagnosis: the stories are simply too large. Two popular fixes don't work:

- "**Keep the average, squeeze the variation**" through better estimating and planning. In real Lead Time data, the spread grows with the average; you can't shrink one without the other.
- "**Make stories half a sprint**", so that even the longest still fits. Lead Times are lopsided, so the long tail spills over anyway.

What works is to **shift the whole distribution left** by making items smaller. His coaching question: *what would it take to deliver twice as many stories over the next few sprints, covering roughly the same work?* Then ask again. His rule of thumb: the **median** story should take no more than about **a fifth** of the sprint, and in extreme cases a tenth.

Do the cheap, risky part first

Reinertsen's perpetual-motion car: **module 1**, the engine, costs **$5 million** with a **5%** chance of working; **module 2**, the rest of the car, costs **$100 million** and is certain to work. Build module 1 first: expected cost $5M + 5% × $100M = **$10 million**. Build module 2 first: always **$105 million**. *Do first the work that removes the most risk for the least money.*

Where to look for large batches

WHERE	LARGE BATCH	SMALLER ALTERNATIVE
Scope	One big release	Fewer features, released more often

WHERE	LARGE BATCH	SMALLER ALTERNATIVE
Funding	Annual approval of a whole programme	Fund in stages as risk falls
Phases	100% of requirements before design	Hand over what's ready
Planning	Detailed plan for the whole project	Detailed plan only for the next weeks
Testing	One big cycle at the end	Continuous, in small pieces
Customer delivery	Large, infrequent deliveries	Smaller, frequent deliveries (*It's Not Luck*)
Lessons learned	One postmortem at the end	Regular retrospectives

WHAT THIS MEANS FOR YOU

- Most batches are **far too large**. Smaller batches cut Lead Time **without new capacity**.
- Smaller batches can make **unit cost look worse** while the business gets better.
- The optimum is **forgiving and usually smaller than you think**: go about 30% below your estimate.
- Make small batches affordable by **cutting transaction cost**.
- Start with **hand-offs**: pass work on as soon as each piece is ready.
- **Do the cheap, risky part first.**
- Make items small enough: a **median** under about a fifth of your planning cycle.

CHAPTER 9

WIP limits, aging and flow debt

A busy signal

When every line is busy, a telephone network gives you a busy signal. It doesn't accept your call and hold it indefinitely. Reinertsen uses this as his example of a **WIP limit**: a cap on how much work a system accepts at once. Turning some demand away at the door is how the calls that *are* accepted keep working.

Toyota did the same with limited space between workstations and the rule "stop when it's full" (Chapter 1). Goldratt does it with the rope of drum–buffer–rope (Chapter 7). Kanban Method does it with limits on columns.

Why limit WIP

By Little's Law, Lead Time follows WIP. Cap the WIP and you cap Lead Time. Chapter 5 showed you can't manage through utilisation, because you can't measure it; Chapter 4 showed Lead Time warns too late. **WIP is the number you can see and control today.**

Reinertsen modelled the trade-off. A **light** limit, about **twice** the usual WIP, cut Lead Time by about **28%** for about **1%** lost in blocked work and idle time. A **heavy** limit, half the usual WIP, lost much of the gain to blocking and idleness. In a worked example, the savings in cost of delay were about **10 times** the cost. (*A model, not field data; the shape is the useful part.*)

> **TEAM ATLAS** averaged about 6.2 items in progress. A light limit would be around **12**: invisible on a normal day, decisive in a bad week.

Vacanti warns against the most common way managers try to get round the limit: **overtime and extra people tend to increase WIP faster than delivery**, so Lead Time gets longer, not shorter.

Zheglov describes a pattern most managers will recognise. A team asks whether it can use its **velocity** as its WIP limit for a sprint. But velocity is a rate (items per sprint) and WIP is a count (items in progress). It's like comparing kilometres with kilometres per hour. If a team starts as many stories as its velocity, Little's Law puts the average story at **a whole sprint**, and with even a little variation, **about half the stories won't finish** by the end. The team commits to ten and delivers five, sprint after sprint. The fix is a WIP limit **well below** the number of stories planned, so each story moves through quickly.

He also offers a useful way to picture a WIP limit: **the kanban is not the card, it's the empty space.** Cards are demand. Empty slots are capacity: permission to start something. Reserving, say, one slot in five for one type of work is a capacity allocation, and it makes some pull decisions automatic.

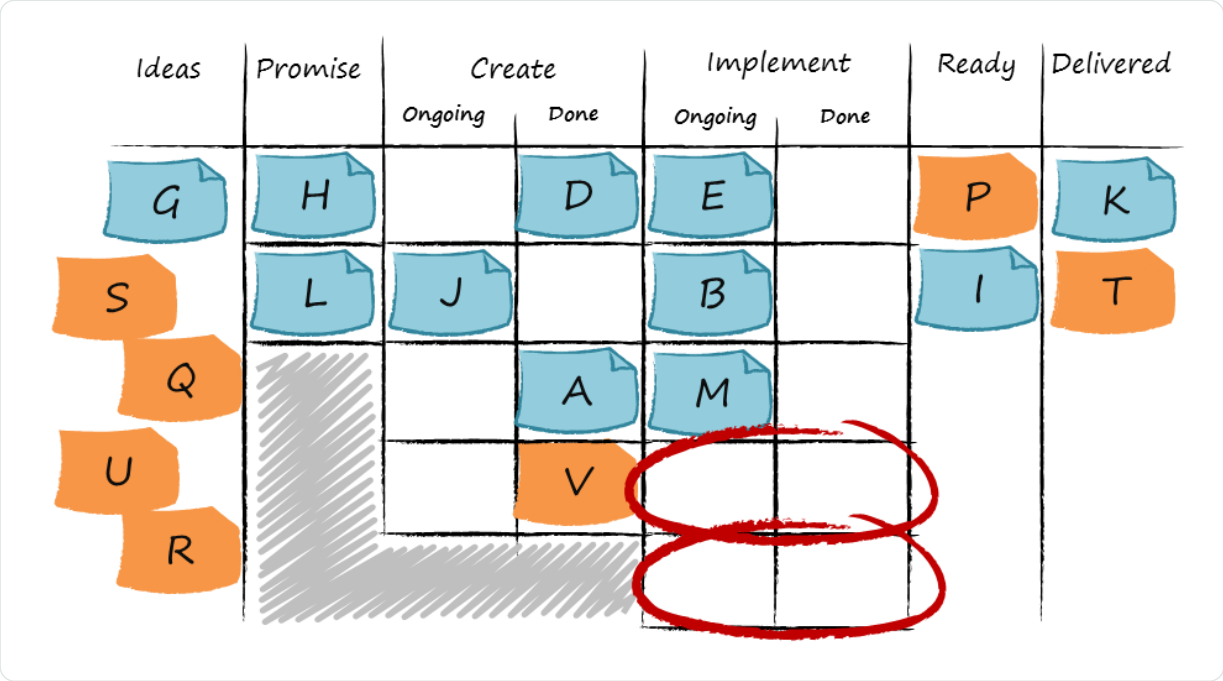


Figure 10 · The kanban is the empty space: free slots are permission to start work

Image: Alexei Zheglov, "Kanban Is Not a Card, It's a Space" (connected-knowledge.com, 2016). The circled empty slots are the kanban: capacity to pull new work.

One limit, or many?

APPROACH	HOW IT WORKS	BEST WHEN
One global limit (drum–buffer–rope)	Release work only as fast as the constraint can absorb	The constraint is stable
Local limits (kanban)	Cap the work between each pair of steps	Bottlenecks move

Reinertsen generally prefers local limits because they respond wherever the problem appears, and fast: in his Toyota example, a kanban signal reaches the previous step in about 3 minutes, around **400 times faster** than a planning system that reacts in about 20 hours.

When you hit the limit

A limit is only useful if something happens when it's reached. Reinertsen's menu:

Reduce demand: block new work; purge low-value work; drop features you decided in advance were optional.

Add capacity, fast: pull in help (even a little helps: a motorway lane blocked for **10 minutes** can cause a **40-minute** jam); use part-time people for unpredictable work; bring in experts, but only if they aren't fully booked; develop people who can work across steps; cross-train neighbouring steps; change the mix of work you start next.

Most of these depend on decisions made **long before** the limit is hit. That's the management work.

Kill the zombies

Many organisations, rather than stop a weak project, quietly **starve** it: too few people to succeed, never formally ended.

> "*Zombie projects destroy flow. Kill the zombies!*" (Reinertsen)

The obstacle is **sunk cost**: "We've already spent €400,000" (*illustrative figure*). Money already spent is irrelevant. The only question is whether the **remaining** investment is worth the **remaining** return.

Hold work where waiting is cheap

> ***Constrain WIP in the section of the system where the queue is most expensive.***
>
> — REINERTSEN

Air traffic control holds planes **on the ground**, not circling at the destination. In knowledge work, the cheapest place for work to wait is **before it is committed**. Once started, it creates hand-offs, partial designs, open questions and status meetings, and it goes stale (Chapter 3).

Watch the oldest items

All schools agree that averages hide problems.

> ***Watch the outliers.***
>
> — REINERTSEN

Reinertsen's analogy is from finance: **accounts receivable** more than 90 or 120 days overdue get special attention, because very late invoices are far less likely to be paid. Old work is the same. The *Kanban Maturity Model* calls aging work **perishable**; Vacanti tracks **work item age** as one of his four basic flow measures.

Vacanti suggests age checkpoints based on the team's own history: when an item reaches the **50th**, **70th** or **85th** percentile of past Lead Times, it's time to ask why. Reinertsen recommends a **preplanned escalation rule**: his example is design drawings waiting more than **7 days**, reported automatically to the department manager.

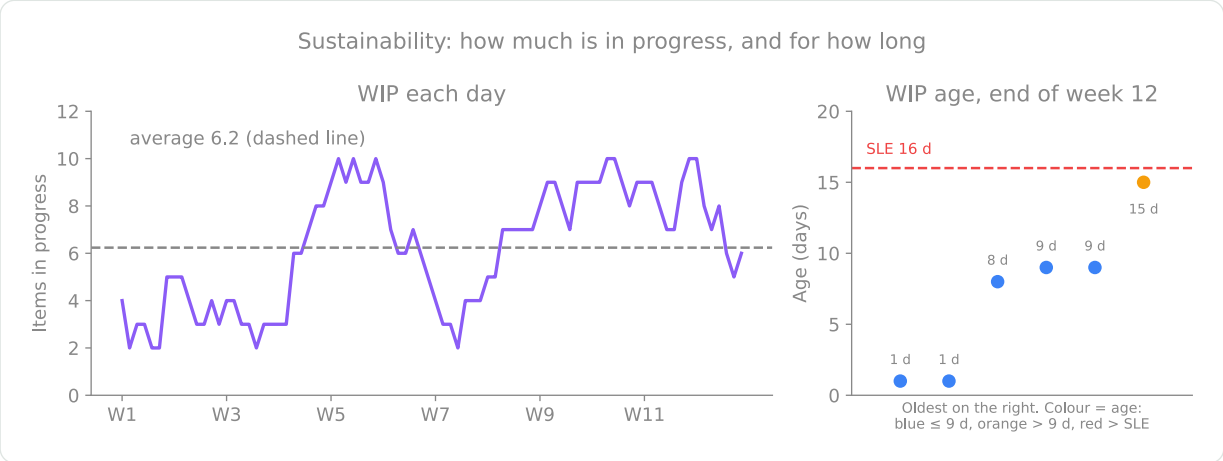


Figure 11 · Team Atlas WIP each day and age of items still in progress

Team Atlas: right-hand panel, the six items in progress at the end of week 12. One has been open 15 days: just under its 85th percentile of 16 days (its SLE, Chapter 11), but above its 70th percentile of 13. With an escalation age set at the 70th percentile, it would already be on someone's desk.

Vacanti also suggests watching **total age**: the sum of the ages of everything in progress. He describes it as the organisation's **total investment in unfinished work**. If it keeps rising, money is being tied up faster than it is being turned into delivery.

Flow debt

Vacanti adds a concept from finance: **flow debt**.

When an item is expedited, pushed to the front ahead of others, it finishes faster. But the time doesn't come from nowhere. It is **borrowed** from the items that were pushed back. They age more than they otherwise would. That borrowed time is flow debt, and it must be repaid: either as longer Lead Times for the delayed items, or by abandoning them, wasting the effort already spent.

Flow debt also violates an assumption of Little's Law: one of its conditions is that the average age of work in progress stays stable. When expediting makes items age artificially, forecasts based on past averages become unreliable.

You can see flow debt in the data: when the CFD suggests one Lead Time but delivered items show longer ones, something is aging more than expected. The most common cause is expediting.

Different limits for different work

Reinertsen and the *Kanban Maturity Model* both suggest **classes** of work with their own limits (Chapter 10). Without limits per class, the system drifts towards everything being urgent. Reinertsen's software maintenance example: **one "hot" job** at a time, at the front of every queue, but not interrupting work already under way.

Small reductions add up

Reinertsen describes an industrial R&D organisation that made its WIP visible and reduced it slowly, finishing slightly more than it started, week after week. Average Lead Time for advanced R&D fell from **220 days to 30 days**.

WHAT THIS MEANS FOR YOU

- A **WIP limit** caps Lead Time. A **light** limit, about twice your usual WIP, gives most of the benefit.
- Overtime and extra people often **raise WIP faster than delivery**.
- **Velocity is not a WIP limit.** Starting as many items as you usually finish means about half won't finish.
- Decide in advance **what happens at the limit**.
- **Kill the zombies.** Ignore sunk cost.
- Let work wait **before commitment**, where waiting is cheap.
- Watch **the oldest items** and **total age**; set escalation points at your own percentiles.
- Every expedite creates **flow debt**: time borrowed from other work, repaid later.
- **Small, steady reductions** work. Make WIP visible.

CHAPTER 10

What goes first: sequencing and classes of service

THE SHORT ANSWER.

Choose by **economics when you commit** to work: cost of delay, WSJF or money per constraint hour. **After commitment, first in, first out** within a few classes of service, with strict limits on urgent work. Shrink queues first; then order matters less.

Four answers to one question

"What should we work on next?" may be the question managers argue about most. The schools in this guide give four different answers:

SCHOOL	RULE	CORE ARGUMENT
Reinertsen	**WSJF**: cost of delay ÷ duration	Minimise the total cost of delay
Goldratt, Anderson, Tendon	**Money per hour of the constraint**	The constraint's time is the scarcest resource
Kanban Maturity Model	**Classes of service** by the shape of cost of delay; first in, first out within a class	Manage risk, not just value
Vacanti	**First in, first out** once work is committed; use value only when choosing what to commit to	Every departure from FIFO hurts predictability

They look incompatible. They aren't, once you see **when** each applies.

Reinertsen: when order matters

Reinertsen's emergency room: patients arrive at random, some need five minutes and some five hours, some can wait and some can't. Nobody runs it first-come, first-served. But when the waiting room is **nearly empty**, nobody bothers with triage either.

> ***Queue cost is affected by the sequence in which we handle the jobs in the queue.***
>
> — REINERTSEN

Order matters when items differ in cost of delay, differ in size, and the queue is long. Then:

- Same cost of delay, different sizes: **shortest first**.
- Same size, different costs of delay: **most expensive delay first**.
- Both different: **WSJF = cost of delay ÷ duration**.

ITEM	COST OF DELAY PER WEEK	DURATION	WSJF
A	€2,000	1 week	2.0
B	€10,000	4 weeks	2.5
C	€6,000	2 weeks	**3.0**

ORDER	CHOSEN BY	TOTAL COST OF DELAY
A → B → C	First come, first served	€94,000
B → C → A	Highest cost of delay first	€90,000
A → C → B	Shortest first	€90,000
C → B → A	**WSJF**	**€86,000**

Reinertsen also rejects two common approaches: ranking by **ROI alone** (a very profitable project may lose little from waiting), and prioritising whatever is **closest to missing its date** (that minimises complaints, not losses).

Goldratt and Anderson: money per constraint hour

If one step is the constraint (Chapter 7), the scarce resource is **its** time. Anderson ranks work by the money it generates per hour of the constraint. Tendon notes that WSJF applies this idea when the constraint isn't known: cost of delay divided by total duration instead of constraint time. Both rank work by **value per unit of the scarcest resource**.

The Kanban Maturity Model: classes of service

Chapter 2 introduced four shapes of cost of delay. The *Kanban Maturity Model* turns each into a **class of service**: a set of rules for how such work is treated.

CLASS	RULE	TYPICAL SHARE
Expedite	Goes first, may exceed WIP limits	Small, strictly limited
Fixed date	Start early enough to make the date with high confidence	As needed
Standard	First in, first out	The largest share

CLASS	RULE	TYPICAL SHARE
Intangible	Has its own reserved share and is the first to give way when expedite work arrives, so it absorbs the peaks	A deliberate reserve (the model's example: 30%, alongside 20% fixed date and 50% standard)

The model is explicit about the price: "*any class of service with policies other than FIFO queuing negatively affects flow and adds delivery risk.*" With several classes, the tail of the Lead Time distribution gets longer and fatter. Classes of service are a **trade-off**: predictability given up for value or risk. The model's safeguards: limit how many items each class may hold, and reserve capacity for the intangible class as a hedge; the more unpredictable the arrival of urgent work, the bigger that reserve should be.

Vacanti: protect predictability

Daniel Vacanti simulated the same team under different rules for picking work. Here is how long 85% of items took:

HOW WORK WAS PICKED	85% DONE WITHIN
Strict first in, first out, no urgent items	50 days
First in, first out, always one urgent item	65 days
Random order, no urgent items	60 days
Random order, always one urgent item	**100 days**

Same team, same work. Combining random order with a permanent urgent item **doubled** delivery time. Vacanti's conclusions: FIFO is the clear winner for predictability; classes of service, used freely, are one of the worst things you can do to it; and the value of a piece of work usually **can't be known reliably in advance**. If you need urgent lanes, you need **slack** to absorb them (Chapter 5).

Zheglov reports a figure Vacanti presented at a conference: allocating **10% of capacity to expedited work roughly doubled Lead Times** for standard work. His image: a motorway closed during rush hour for a presidential motorcade. When will the commuters get to work? He also stresses that an expedite lane isn't magic. The point is to **reduce** urgent work, not to handle it forever. In the Corbis case that Anderson describes, expedites were cut to about **five a year** in a company of around 1,300 people.

Putting them together

The disagreement is mostly about **timing**: *when* in the life of an item economics should decide the order.

1. **Make queues small first** (Chapters 4, 8, 9). With short queues, order matters little, and FIFO works well.

2. **Use economics at the commitment point.** When choosing what to start next, use cost of delay, WSJF or money per constraint hour.
3. **After commitment, use first in, first out within a few classes.** Don't reshuffle work already in progress. This protects predictability and limits flow debt (Chapter 9).
4. **Use a small number of classes, with limits.** Expedite, fixed date, standard, intangible is enough. Limit expedite tightly and keep capacity for the intangible class.
5. **Make the rules visible.** Priorities based on **economic facts** are more stable than priorities based on opinion, which shift with whoever spoke last.

STEP	WHO BACKS IT
1. Small queues first	Reinertsen, Vacanti
2. Economics at commitment	Reinertsen, Goldratt, Anderson; Vacanti accepts it here
3. FIFO after commitment	Vacanti
4. Few classes, with limits	*Kanban Maturity Model*
5. Visible rules	Reinertsen, *Kanban Maturity Model*

Goldratt's *The Goal* ends up in the same place. The plant first uses red and green tags to show priorities everywhere. Later, once work is released at the constraint's pace and the buffers are right, the team finds the tags are no longer needed: **plain first in, first out works**.

Keep it simple

Reinertsen is wary of elaborate scoring models that produce precise priorities from imprecise inputs. The aim is to **avoid the big mistakes**: a €50,000-a-week item stuck behind a €2,000 one. For work whose duration is unknown, use **time slices**: each item gets a fixed slice; if it isn't finished, it goes to the back. And for urgent work, prefer **head of the line** (it goes first in the queue) over **interruption** (someone stops what they're doing), because interrupted work ages too.

WHAT THIS MEANS FOR YOU

- Order matters most when **queues are long**. Shrink queues first.
- **At commitment**, choose by economics: WSJF or money per constraint hour.
- **After commitment**, first in, first out within a **few classes** of service.
- Every exception to FIFO **costs predictability**. Limit expedite, and keep slack.
- Don't rank by **ROI alone** or by **who is closest to their date**.
- Base priorities on **economic facts**. Aim to avoid big mistakes, not to rank perfectly.

CHAPTER 11

Promises and forecasts as economic tools

The weather forecast

Imagine a forecast that said "Rain tomorrow", never "70% chance of rain". You'd soon stop trusting it. Most delivery promises are like that: "It'll be done on 15 June." No range, no confidence. When it slips, trust drops.

This chapter is about promises: how to make them honestly, what they cost, and how the schools differ.

A promise is a range and a probability

Vacanti's rule: **a proper forecast has two parts, a range and a probability.** "85% of our items are delivered within 16 days." This is a **service level expectation** (SLE). Agreed formally with a customer, it becomes a **service level agreement** (SLA).

Vacanti's earlier training material proposes a promise for each class of service:

CLASS	PROMISE
Expedite	As soon as possible, with full transparency
Fixed date	On time, with enough notice to start early
Standard	Within the 85th percentile of past Lead Times

The economics of confidence

Why 85%? Troy Magennis, who made it the default in his widely used forecasting spreadsheets, admits it has **no mathematical basis**. It just felt like a balance between the 50th percentile, which is a coin toss, and the 95th, which people found too pessimistic.

The real point is economic: **the confidence level is a business decision**, set by the cost of being late.

- If lateness is cheap, a lower percentile is fine.
- If lateness is expensive (a contract penalty, a regulatory date), go higher: 90%, 95%, even 99%.
- The team provides the range; the person accountable for the risk chooses the level.

Vacanti adds the other side: **over-conservative promises cost money too**: missed market opportunities, lost customer confidence, plans built around dates later than necessary. The *Kanban Maturity Model* uses both the 85th and **98th** percentiles: the first for the normal promise, the second to understand the worst realistic case.

And remember Vacanti's warning from Chapter 6: a forecast from an **unstable** process can't be trusted, whatever percentile you choose.

One promise doesn't fit all work

Zheglov shows why a single SLE for "all our work" can mislead. In one IT services organisation, all items together averaged **34 days**, with 85% within **61** and 98% within **96**. Split by type of work, the picture changed:

WORK TYPE	AVERAGE	85% WITHIN	98% WITHIN
Projects	30	54	85
Reports	35	61	96
Implementations	40	66	81
Integrations (phase-gated)	56	70	100

Days. Zheglov's case data.

Each type deserves its own promise. He adds a warning about ambition: on typical Lead Time shapes, **guarantees beyond about 80% become very expensive**, because the tail stretches far beyond the average. That doesn't rule out the 85th or 98th percentile; it puts a price on them. Pay it when the cost of being late is higher than the cost of the longer promise, as with a fixed regulatory date. For everyday work, a lower percentile is often the better deal.

Predictability is worth money to someone else

Zheglov tells the story of Jack, who was planning a new project that depended on two other teams. One team measured its flow and could say: **90% of requests within 20 days**. Jack could build his own promises on that. The other could only say that his request was "in the backlog, among hundreds". Jack couldn't rely on them, and had to choose a **more expensive** make-or-buy option.

The point for managers: your team's predictability is part of **other teams'** economics. Ask who depends on you, what they expect, and whether you measure what they need.

Forecasting a batch of work: Monte Carlo

For "when will these 40 items be done?", Vacanti and Magennis recommend **Monte Carlo forecasting**. Instead of dividing remaining work by an average pace, the computer replays the team's real weekly history thousands of times: picking a random past week, then another, until the work is done. Counting the outcomes gives dates with probabilities.

For **Team Atlas**, with 40 items left and its 12 weeks of history:

CONFIDENCE	DONE WITHIN
50%	9 weeks
70%	10 weeks
85%	**11 weeks**
95%	12 weeks

The traditional "average pace" answer was 9 weeks: a coin toss nobody admitted to. Re-run the forecast weekly; if the 85% date starts slipping, you see it early.

The *Kanban Maturity Model* adds a limit: Monte Carlo is most useful in the middle ground. With very stable, thin-tailed data, simpler methods may do; with fat tails, no forecast is reliable until the tail is shortened (Chapter 6).

Goldratt: one buffer, visible to everyone

Goldratt's *Critical Chain* approaches promises from the project side:

1. **Remove hidden safety from each task.** Estimate tasks closer to the likely time than to the safe time.
2. **Put the safety in one place**: a **project buffer** at the end, protecting the delivery date. Add smaller **feeding buffers** where side paths join the main chain.
3. **Plan around resources, not just tasks.** If one person or team is needed on several paths, the longest chain through the project is set by them, not just by task order. Goldratt calls this the **critical chain**.
4. **Manage the buffer, not the tasks.** Track how much of the buffer has been used against how much of the work is done.

Steve Tendon builds this into a simple signal. **Buffer burn rate** = share of buffer used ÷ share of work completed. If 40% of the buffer is gone but only 20% of the work is done, the burn rate is **2.0**: trouble ahead. A **fever chart** plots this over time in three zones:

ZONE	MEANING	ACTION
Green	Variation as expected	Leave it alone
Yellow	Normal but noticeable variation	Plan a response

ZONE	MEANING	ACTION
Red	Abnormal variation	Act now

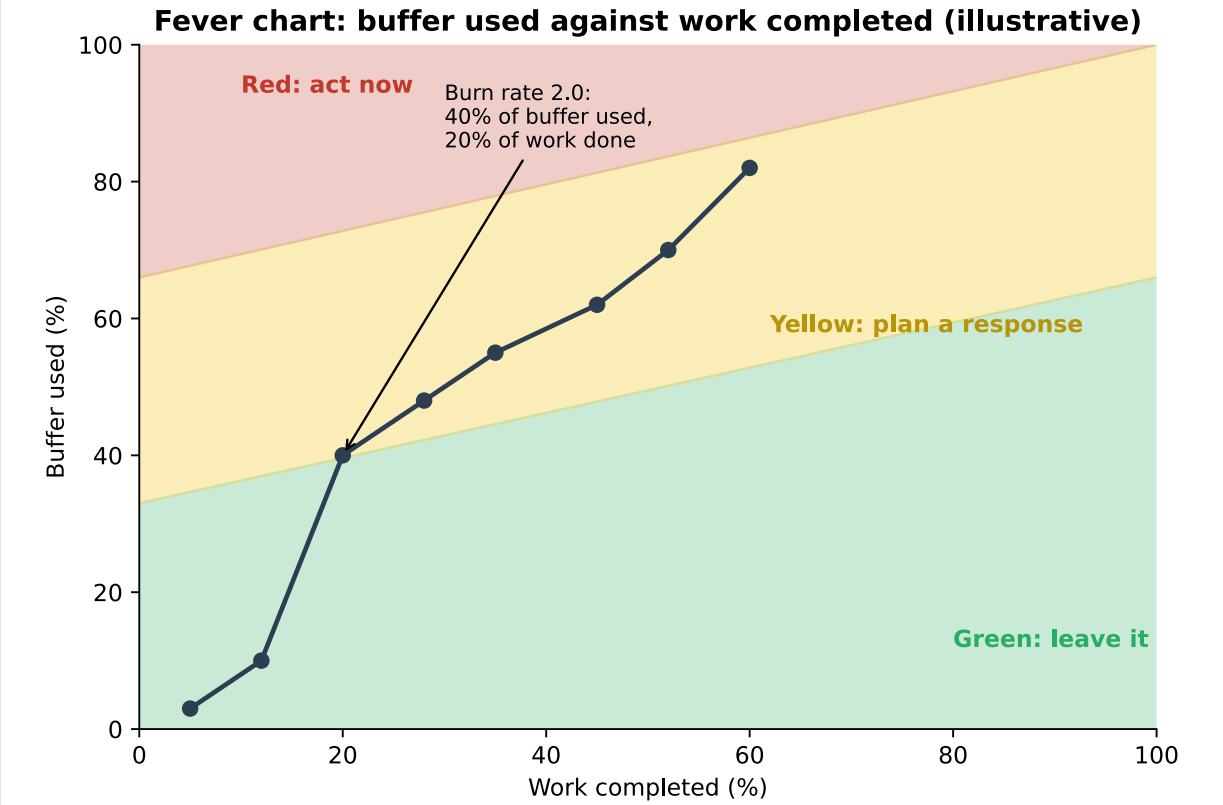


Figure 12 · Fever chart: buffer used against work completed, in green, yellow and red zones

Tendon suggests sizing buffers from the team's own Lead Time data, around the 60th to 85th percentile, rather than from guesses. He also uses the buffer colour to decide on **extra work**: green, accept it; yellow, judge; red, refuse.

Three views reconciled

Reinertsen's warning from Chapter 6 still applies: "A *buffer converts uncertain earliness to certain lateness*." How do the three views fit?

QUESTION	ANSWER
Should every task carry its own safety?	**No.** All three schools agree
Should a promise include a margin?	**Yes, made visible and chosen deliberately.** A percentile (Vacanti) or a shared buffer (Goldratt)
How big should it be?	**Set by the cost of being late** (Reinertsen, Magennis), sized from the team's own data (Vacanti, Tendon)

QUESTION	ANSWER
How do we know if we're in trouble?	**Leading signals**: item age against percentiles (Vacanti), buffer burn rate (Goldratt, Tendon)

Promises and trust

The *Kanban Maturity Model* makes an observation managers should hear: **deadlines imposed without data are an instrument of low trust.** Vacanti adds that an SLE should be **set from data, with the customer**, never imposed by management from outside, and never used to punish teams. A promise used as a weapon produces gamed data, padded estimates and, in the end, worse delivery.

WHAT THIS MEANS FOR YOU

- Every promise needs **a range and a probability**.
- The **confidence level is a business decision**, set by the cost of being late. 85% is a convention, not a law.
- Over-conservative promises **cost money too**; guarantees beyond about 80% get expensive.
- Set promises **by type of work**, and remember that others build their plans on yours.
- Use **Monte Carlo** for batches of work; re-run it weekly.
- Don't hide safety in tasks. Use **one visible buffer** or a **percentile**.
- Watch **leading signals**: item age, buffer burn rate, fever charts.
- **Never use promises to punish.** Set them from data, with the customer.

CHAPTER 12

Cadence and feedback

The school bus

Reinertsen rode a bus to school in New York. Buses arrived in bunches: twenty minutes with none, then three at once. A bus a little behind finds more passengers, loads more slowly and falls further behind; the one behind it catches up. The cure: **fixed departure times**. An early bus waits. Each delay stays inside one interval.

That is **cadence**: doing things at regular, predictable intervals.

What cadence does

EFFECT	REINERTSEN'S PRINCIPLE
Stops small delays building up	Reset regularly, and reset to the middle, not just inside the limit
Makes waiting predictable	A release missed this month catches next month's. Without cadence, every stakeholder crams into the current release, which grows and slips
Makes small batches cheap	A fixed time means no set-up cost for the meeting
Speeds decisions	A daily meeting responds within 24 hours; gathering decision-makers ad hoc can take 72 hours

He adds that, in his experience, *teams that meet daily spend no more time in meetings than teams that meet weekly*, because less piles up between meetings.

Cadence has one condition, linking straight back to Chapter 5: **it needs spare capacity**. A team at 98% can't keep a rhythm. When time is tight, Reinertsen's advice is to let **scope** flex, not the date.

The cadences of a kanban system

The *Kanban Maturity Model* describes a set of regular meetings, each with its own purpose and rhythm:

CADENCE	PURPOSE
Daily meeting	Look at the board: what is blocked, what is aging, what needs help
Replenishment	Choose what to commit to next. This is where economics decides (Chapter 10)

CADENCE	PURPOSE
Delivery planning	Decide what is released, and when
Service delivery review	Compare delivery with what customers expect; adjust policies
Operations review	Look across teams and services; find dependencies and shared problems
Risk and strategy reviews	Less frequent; look at the portfolio and the direction

How often should each happen? The model's rule: choose the frequency by balancing **the cost of meeting** against **the cost of the delay before the next decision**. If the information changes quickly and decisions are expensive to delay, meet often; if not, less often. Vacanti adds a practical point: replenishment and delivery **don't have to share the same rhythm**. Staggering them lets each follow its own economics.

Synchronisation

Cadence means regular intervals; **synchronisation** means doing things **at the same time**. Reinertsen describes a company whose engineering changes passed through **seven reviewers**, one after another. Any could send a change back. Replacing that with a **weekly meeting of all seven together** cut average processing time from **40 days to 5**.

Most organisations have a version of this: a change board, an architecture review, a security sign-off, each done separately. Putting them in one room at one regular time can cut weeks out of every change.

Nest the rhythms

> ***Make nested cadences harmonic multiples.***
>
> — REINERTSEN

Daily inside weekly, monthly inside quarterly, quarterly inside yearly: each longer rhythm a whole multiple of the shorter one. A week doesn't fit evenly into a month, so pair weekly rhythms with two- or four-week cycles rather than calendar months. When rhythms nest, they reinforce each other; when they don't (a three-week cycle inside a monthly release), they drift apart and create waiting at every mismatch.

Fast feedback

Reinertsen's favourite sailor's saying: "A *little rudder early is better than a lot of rudder late.*" When feedback is slow, managers react too late and too hard. When it is fast, small corrections are enough.

In knowledge work, fast feedback has two jobs: **cut losses early** (stop a bad path before much is spent) and **catch opportunities** (spot something better and move to it). Fast feedback and small queues reinforce each other: less WIP means shorter waits, shorter waits mean faster feedback, faster feedback means less rework and less WIP.

Vacanti adds a discipline for using feedback: **react to signals, not noise** (Chapter 6). A feedback system that triggers investigation every time a number wobbles creates more instability, not less.

TEAM ATLAS. Atlas's delivery swung between 0 and 8 items a week. A fixed weekly review of WIP, oldest items and new requests, plus a separate replenishment meeting where new work is chosen by cost of delay, would give it a regular point to reset: decide what to finish, what to start and what to hold back.

WHAT THIS MEANS FOR YOU

- **Cadence** stops small delays building up and makes waiting predictable.
- It makes **small batches cheap** and **decisions faster**.
- It **needs spare capacity**. When time is tight, flex scope, not the date.
- Separate the **cadences**: daily, replenishment, delivery, service review, operations review. Set each by the cost of meeting versus the cost of waiting.
- **Synchronise** reviewers: one joint meeting can replace weeks of hand-offs.
- **Nest** the rhythms.
- Fast feedback **cuts losses and catches opportunities**. React to signals, not noise.

PART IV

Managing the system

CHAPTER 13

Metrics that show the economics

What gets measured may not get done

Reinertsen questions a famous management saying, "*What gets measured gets done.*" His example is a bathroom scale: weighing yourself every day doesn't make you lose weight. A metric is only the **measurement** part of a control system. Without someone who acts on it, a rule for when to act, and feedback fast enough to matter, it changes nothing.

Goldratt states the other half: "*Tell me how you measure me, and I'll tell you how I'll behave.*" Metrics don't just describe behaviour. They create it. A metric is worth having only if it's clear **who acts on it, when, and how**, and if the behaviour it encourages is the behaviour you want.

Four families of metrics

Each school brings its own measures. They answer different questions, and a manager needs a few from each.

FAMILY	MEASURES	QUESTION	MAIN SOURCES
Flow	WIP, Lead Time, delivery rate, work item age	How is work moving?	Vacanti
Queues and causes	Total WIP and its trend, oldest items, batch size, transaction cost, feedback speed, decision time	Why is it waiting?	Reinertsen
Money	Throughput (T), investment (I), operating expense (OE), net profit, return; cost of delay; cost of queues	What is it worth?	Goldratt, Anderson, Reinertsen
Risk and promises	Percentiles, SLE attainment, buffer burn rate, share of expedited work	Can we keep our promises?	Vacanti, Magennis, *Kanban Maturity Model*, Tendon

Rules for using them

The schools agree on a handful of rules.

1. **Count items; don't estimate effort.** (WSJF in Chapter 10 needs only a rough size, such as S/M/L or the median Lead Time for that type of work, not a detailed estimate.) Reinertsen calls counting items in a queue "*astonishingly effective*". Software teams have counted open bugs for decades, even though bugs vary enormously in effort: estimating each one would take almost as long as fixing it. Count what the customer values.

2. **Measure total WIP**, not just what people call "waiting" (Chapter 4). Reinertsen reports companies achieving **5 to 8 times** shorter Lead Times simply by measuring and driving down total WIP.
3. **Prefer leading indicators.** WIP, queue trends, item age and buffer burn rate warn early. Lead Time and delivery rate confirm late.
4. **Use percentiles, not averages and sigma**, for anything you promise (Chapters 1, 6, 11). "Average ± 3 sigma" control limits are wrong for lopsided data.
5. **Separate signal from noise** before reacting (Chapter 6).
6. **Set triggers in money, not days.** A flat rule such as "any project more than four weeks late goes on the watch list" treats a €10,000 slip and a €100,000 slip the same. Reinertsen's alternative: decide what size of problem is worth acting on, say €50,000, and set each project's trigger at the deviation that would cost that much.
7. **Use utilisation only for long-term planning**, never for weekly control (Chapter 5).
8. **Keep delivery rate (items) and throughput (money) apart** (Chapter 3).

Decide before you measure

Zheglov, quoting the decision researcher Douglas Hubbard, turns the usual order around: **decide first what you will decide, and how**, then collect data. "*Your most important decision is the decision-making method that you're going to use.*" If data comes first, organisations collect too much, spend too much getting it, and people start to fear what it will be used for, so they bring different "data". If the decision comes first, people find just the data it needs.

He gives a related warning about scores. On a 0–10 satisfaction scale, "*their eight is not another participant's eight*": people calibrate differently. A short set of described levels (exceeded expectations, fully met, mostly met with minor concerns, significant concerns…) is more reliable than a number.

The one-slide review

David Anderson suggests reporting to senior management on **one slide** a month: net profit and return on investment at the top, a few production measures beneath. Combining his idea with the other schools gives a slide like this:

SECTION	MEASURES
Money	Throughput generated (or estimated value delivered), operating expense, investment tied up in accepted-but-undelivered work; cost of delay of the main items waiting
Flow	Total WIP and trend; delivery rate; Lead Time at the 50th and 85th percentiles
Risk	SLE attainment; oldest items against escalation age; buffer status or fever chart for key deliveries; share of work expedited
The constraint	Where it is now; how it was protected and exploited this month; any case for elevating it

SECTION	MEASURES
Decisions needed	What management is asked to decide, with the money attached

The last row is the point of the exercise. Reinertsen: to influence financial decisions, **speak the language of money**.

A starter panel for team managers

For a team or service, a smaller panel is enough. Most of it needs no new tools.

MEASURE	LOOK AT IT	ACT WHEN...
Total WIP	Weekly	It keeps rising, or exceeds the agreed limit (Chapter 9)
Age of oldest items	Weekly	An item passes the agreed escalation age
CFD	Weekly	The arrival and delivery lines move apart
Lead Time, 50th and 85th percentiles	Monthly	The 85th percentile creeps up, or points fall outside routine variation
Share of work expedited	Monthly	It grows; check for flow debt
Cost of delay of the main items	When priorities change	Two items' order looks wrong economically (Chapter 10)
Cost of the biggest queue	Quarterly	It's large enough to justify investment in capacity or automation
Batch size and transaction cost	Quarterly	Releases, reviews or approvals are getting bigger or rarer

TEAM ATLAS would show: about 6 items in progress, the oldest at 15 days against an 85th percentile of 16; arrivals slightly ahead of deliveries overall (55 started, 53 finished); Lead Time median 9, 85th percentile 16; and, once a cost of delay is agreed, a weekly price on the waiting.

Metrics that do harm

Several sources list measures that look sensible and damage flow:

METRIC	WHY IT DOES HARM	SOURCE
Utilisation or "busyness" per person or department	Drives overload and long queues	Reinertsen, Goldratt, *Kanban Maturity Model*
Local efficiency or productivity	Creates local optima (the robots' +36%)	Goldratt
Cost per unit	Rewards big batches	Goldratt, Anderson

METRIC	WHY IT DOES HARM	SOURCE
Story points or velocity as targets	Easy to inflate; mean little outside the team	Vacanti
Arbitrary targets ("lead time under 10 days")	Teams hit the number by gaming the data	Vacanti
Comparing this week with last week	Treats noise as signal	Vacanti
Metrics that divide responsibility perfectly	*"Mathematically elegant"* but *"bad psychology"*: nobody helps anyone else	Reinertsen

WHAT THIS MEANS FOR YOU

- A metric matters only if it's clear **who acts on it, when and how**. Metrics shape behaviour.
- Use four families: **flow, queues and causes, money, risk and promises**.
- **Count items**, measure **total WIP**, prefer **leading indicators**, use **percentiles**.
- Set **triggers in money**, not in days.
- **Decide what you'll decide** before collecting data.
- Report upwards on **one slide**, ending with the decisions needed.
- Stop using **utilisation, local efficiency and arbitrary targets** as management measures.

CHAPTER 14

Management as the constraint

"Because that's how you manage"

Imagine a task needing four person-days of work that took half a year to deliver. The manager asked: *"Why on earth so long?!"* The answer: *"Because that's how you manage."*

Every school in this guide reaches the same conclusion by a different route. This chapter collects them, because for most organisations, **the biggest constraint on flow is not a team or a machine. It is how the system is managed.**

Policy constraints

At the end of *The Goal*, Alex Rogo's team discovers that the plant's root constraint was not a machine at all, but **management policy**: rules about batch sizes, measures and release of work that everyone followed because they always had. Goldratt calls these **policy constraints**. They cost nothing to remove, and are the hardest to see, because they look like "how things are done".

Typical policy constraints in knowledge work:

POLICY	WHAT IT DOES TO FLOW
Annual budgets approved in one go	Swamps the system with a year's work at once (Anderson)
Every request accepted on arrival	WIP grows; Lead Times stretch (Chapters 4, 9)
People measured on utilisation	Overload; no slack; queues (Chapter 5)
Phase gates handing over 100% of the work	The largest possible batch (Chapter 8)
Approval boards meeting monthly	Decision queues; large batches
Priorities reshuffled weekly by whoever shouts loudest	Flow debt; lost predictability (Chapters 9–10)
Fixed scope with fixed dates, set without data	Padding, hidden buffers, gaming (Chapters 6, 11)

Structures that slow everything

- **Matrix management.** An employee with two managers, functional and project, often leads to chaos and a huge increase in the time of **all** projects. One countermeasure: an internal labour exchange, where an engineer, once assigned, reports only to the project.

- **Indispensable people.** "We can't stop interrupting him; nobody else can do it." An indispensable person may be highly productive locally but destabilises the whole flow. Remedies: shared ownership of code, standards, cross-training. If a person has become indispensable, fire him immediately.
- **Long chains.** The more hand-offs, the more fragile the flow (Chapter 1). Reducing the number of steps can change the picture radically, even if it costs more in direct effort.

Measures that create the problem

Metrics create behaviour (Chapter 13). David Anderson argues that the **cost-accounting mental model**, with its focus on unit cost and utilisation, is what drives organisations into large batches and growing WIP. Reinertsen tells the story of a tooling shop manager measured on **efficiency**: he kept the shop fully loaded and avoided overtime, and tooling kept delaying projects. When his measure became **how long tooling spent on the critical path**, his behaviour "*changed completely.*"

> ***When response time is important, measure response time.***
>
> — REINERTSEN

Daniel Vacanti adds three warnings about how managers use data:

- **Arbitrary targets** (Chapter 13) make people hit numbers, not improve systems.
- **Mandatory overtime distorts the data** that forecasts depend on, and raises WIP faster than delivery.
- **Promises must not be used to punish.** An SLE set from data with the customer improves trust; an imposed deadline used as a weapon destroys it.

Why managers resist

The *Kanban Maturity Model* is unusually honest about why managers don't adopt flow thinking, even when the economics are clear:

- **WIP limits look like underutilisation.** Limiting work feels like wasting people.
- **Saying no to customers** feels like failing them.
- **Martyrdom.** At the middle maturity levels, people keep customers happy by heroic effort: overtime, firefighting, personal sacrifice. It works for a while, it is admired, and it hides the fact that the system is losing money.
- **Siloed metrics.** Each department is measured on its own utilisation, so nobody owns the end-to-end flow.
- **Emotional resistance.** The model quotes Goldratt's view that emotional resistance can itself be the constraint.

For a change agent, this list is a map of where conversations will get difficult.

Decentralise what ages badly

Reinertsen's answer to slow decisions is not "decentralise everything" but a rule:

DECIDE LOCALLY	DECIDE CENTRALLY
Problems and opportunities that **age badly**	Problems that are **rare, large, or have economies of scale**
Give teams the **authority and resources**, not just the responsibility	Measure central teams on **response time**, not efficiency

To keep people aligned without central control, he draws on military practice:

- **Say why, not just what.** Every goal should come with its purpose ("commander's intent"), so people can act sensibly when the plan stops fitting.
- **Define boundaries clearly.** *"If you are going to worry about it, I won't."*
- **Name the main effort.** Of 200 features customers might ask for, only four to six usually change what they buy.
- **Give the rule, not the decision.** The Boeing 777 rule ($300 per pound saved) let 5,000 engineers decide consistently without asking (Chapter 2).
- **Hold some reserve at every level**, not just at the top.

> *"An imperfect decision executed rapidly is far better than a perfect decision that is implemented late."* (Reinertsen)

The *Kanban Maturity Model* reaches the same place through **explicit policies**: write down how work is selected, limited, prioritised and escalated, so that decisions follow agreed rules rather than whoever happens to be in the room.

Trust

Steve Tendon's TameFlow puts **trust** at the centre: a community of trust and a shared purpose, with decisions delegated to the people who know the work. He criticises budgeting, cost accounting and hard targets as structures that destroy trust.

Reinertsen closes the loop with the rest of this guide:

> *"Small batch sizes build trust, and trust enables the decentralized control that enables us to work with small batch sizes."* (Reinertsen)

A team that delivers something useful every month, reliably, builds trust far faster than one that delivers one big thing every few years. Predictability (Chapter 11) is not just a statistical property. It is the basis of trust between teams, managers and customers.

Changing the system without a big batch

Zheglov points out an irony in how organisations try to improve. A typical six-month coaching contract runs to 90–120 billable days and well over **$100,000**, a large batch, sold to teach teams to deliver small batches every two weeks.

He describes a different route at a video software company of over 500 people, where quality was a visible problem. Around 100 people received training, and 27 of them, about 5% of the company, including the CEO, the CFO, the head of engineering and the managers of key departments, were trained to manage flow. The whole engagement took about **25 consulting days**. The effect spread far beyond engineering; the executives named the **accounting** department as one of the places where it had the most impact. The CEO's verdict a year later is worth quoting to anyone working on quality:

> *"Kanban didn't solve our quality problems. It consistently led us to better decisions, which, in turn, led to improvements in quality."*

The lesson for change agents: **train the people who make decisions**, in small batches, and let better decisions improve quality, rather than trying to install quality directly.

The manager's new job

INSTEAD OF...	DO THIS
Keeping everyone busy	Keep **work** moving; protect slack
Accepting every request	**Limit WIP**; choose at the commitment point
Approving every decision	Set **economic rules**; share the cost of delay and the intent
Measuring utilisation and unit cost	Measure **flow, queues and money**
Reshuffling priorities weekly	**FIFO within a few classes**; decide at replenishment
Imposing deadlines	Agree **promises from data**, with the customer
Adding people to late projects	Find and protect the **constraint**
Rewarding heroics	Fix the **system** that needs them

WHAT THIS MEANS FOR YOU

- For most organisations, **management is the main constraint** on flow.
- Look for **policy constraints**: rules everyone follows because they always have.
- Watch for **matrix management, indispensable people and long chains**.
- **Measures create behaviour.** Measure central teams on **response time**, not efficiency.
- Expect resistance: **WIP limits look like idleness; heroics look like commitment.**
- Improve through **better decisions**, by training decision-makers in small batches.
- Decentralise what **ages badly**; keep alignment through **intent, boundaries and rules**.
- **Small batches build trust; trust enables decentralisation.**

CHAPTER 15

Cheat sheet, where to start, glossary

The guide on one page

CHAPTER	THE BELIEF TO DROP	THE IDEA TO ADOPT	THE QUESTION TO ASK
0. Proxies	"Our numbers show how we're doing"	Translate proxies into money; beware local optima	"What does this do to profit?"
1. Flow	"Delay comes from slow work"	WIP and Lead Time are twins; flows have shapes	"What shape is our flow, and where does it wait?"
2. Cost of delay	"Time is free until the deadline"	Quantify a week of delay and its shape	"What does a week of delay cost?"
3. Throughput (money)	"Cut cost first"	T before I before OE; the constraint hour	"Where does the system make money?"
4. Queues	"Queues are a fact of life"	Queues are invisible inventory and a leading indicator	"How much is waiting, and what does it cost?"
5. Utilisation	"Idle people are waste"	Slack makes promises reliable	"Is it cheaper to add capacity or pay for the queue?"
6. Variability	"All variation is bad"	Judge by payoff; reduce consequences; noise or signal	"Is this a signal?"
7. Constraints	"Speed everything up"	Identify, exploit, subordinate, elevate	"What limits the whole?"
8. Batch size	"Big batches are efficient"	Smaller batches, cheaper transactions	"What could we hand on sooner?"
9. WIP limits	"Start everything now"	Limit WIP; watch age; avoid flow debt	"What will we stop starting?"
10. Sequencing	"Biggest ROI first" or "loudest first"	Economics at commitment; FIFO after	"Whose value suffers most from waiting?"
11. Promises	"Give a date"	A range and a probability; one visible buffer	"How confident do we need to be, and what does that cost?"
12. Cadence	"Meet when needed"	A regular rhythm; fast feedback	"When is the next fixed opportunity?"
13. Metrics	"What gets measured gets done"	Metrics with owners and actions	"Who acts on this, and when?"

CHAPTER	THE BELIEF TO DROP	THE IDEA TO ADOPT	THE QUESTION TO ASK
14. Management	"The team is the problem"	The system is the problem	"Which of our policies is the constraint?"

Seven things to remember

1. **Put a price on time.** If you quantify only one thing, quantify the cost of delay.
2. **Most delivery time is waiting.** WIP and Lead Time are twins.
3. **Full utilisation is expensive.** Slack makes promises reliable.
4. **Find and protect the constraint.** An hour lost there is lost for the whole system.
5. **Make batches smaller and limit WIP.** The cheapest levers, needing no new people.
6. **Promise with a range and a probability.** Decide the confidence level by the cost of being late.
7. **Fix the system, not the people.** Most constraints are management policies.

Where to start on Monday

You don't need a programme, a tool or a budget.

1. **Run the 60-day test** on one important piece of work (Chapter 0).
2. **Agree a rough cost of delay** and its shape, with the same people (Chapter 2).
3. **Sketch the flow.** How many hand-offs? What shape is it? Where does work pile up? (Chapters 1, 7)
4. **Count total WIP** and note the five oldest items (Chapters 4, 9).
5. **Price one queue**: the one everyone complains about (Chapter 4).
6. **Set a light WIP limit and an escalation age**, and decide now what happens when either is hit (Chapter 9).
7. **Fix a rhythm**: a short weekly review, and a separate replenishment meeting where new work is chosen by cost of delay (Chapters 10, 12).
8. **Write down one decision rule** and one policy you will stop (Chapters 2, 14).

After a month, look at what changed: WIP, the age of the oldest items, how long decisions took, and how much shorter the arguments became.

Glossary

TERM	PLAIN MEANING
Aging analysis	Looking at how long each item in progress has been open, with attention to the oldest
Batch size	How much work is bundled before being passed on, checked, released or decided

TERM	PLAIN MEANING
Buffer	Deliberate margin: of time (in a plan) or of work (in front of a constraint)
Buffer burn rate	Share of a buffer used ÷ share of the work completed. Above 1 means trouble ahead
Cadence	Doing something at regular, predictable intervals
CFD (cumulative flow diagram)	Chart of running totals of work arriving and leaving. Vertical gap = WIP; horizontal gap ≈ Lead Time
Class of service	A set of rules for how a type of work is treated: expedite, fixed date, standard, intangible
Commitment point	The moment the team agrees to do an item; the Lead Time clock starts
Congestion collapse	When a system is so overloaded that its output falls, not just slows
Constraint	The step that limits how much the whole system can deliver
Cost of delay	The money lost for each unit of time (usually a week) that something is late
Critical chain	The longest chain of dependent tasks in a project, taking shared resources into account
Delivery rate	Items reaching ready for delivery per week (what some authors call throughput)
Drum–buffer–rope	Release work at the constraint's pace (drum), keep a small buffer in front of it, tie release to it (rope)
Fat tail	A Lead Time distribution where a few extreme items dominate; averages become meaningless
Fever chart	A chart of buffer used against work completed, in green, yellow and red zones
FIFO	First in, first out
Five focusing steps	Identify, exploit, subordinate, elevate, repeat
Flow	The movement and delivery of customer value through a process (Vacanti). Lead Time measures how long it takes; flow describes how the work moves
Flow debt	Time borrowed from other work when an item is expedited; repaid through the longer Lead Times of the other work
Flow efficiency	Share of Lead Time during which someone actually worked on the item
Flow types (I, A, V, T)	Shapes of flow: straight line, converging, diverging, late diverging
Investment (I)	Money tied up in things intended to deliver value; in knowledge work, mainly accepted ideas and requirements
Kanban	A way of managing work by making it visible and limiting work in progress at each step
Lead Time	Calendar days from commitment to ready for delivery

TERM	PLAIN MEANING
Leading indicator	A measure that warns of future problems (queue size, item age). Lead Time is lagging
Little's Law	Average Lead Time = average WIP ÷ average delivery rate
Local optimum	An improvement in one part that doesn't improve, or harms, the whole
Monte Carlo forecasting	Replaying real delivery history thousands of times to get dates with probabilities
Multitasking	Switching between several items in progress; stretches each one's Lead Time
Operating expense (OE)	Money spent running the delivery system: salaries, tools, facilities
Parkinson's law	Work expands to fill the time available
Percentile	"X% of items finished within this value"
Policy constraint	A management rule that limits flow
Protective capacity	Spare capacity at steps that aren't the constraint, so they can catch up and keep it supplied
Proxy variable	A measure that stands in for profit, such as utilisation or conformance to plan
Queue	Work that has arrived but is waiting for a person, team, environment or decision
Routine vs exceptional variation	Ordinary noise vs a real change with a specific cause
Sigma (standard deviation)	A measure of spread around the average; "± 3 sigma" limits fit only bell-shaped data
SLE / SLA	A delivery promise with a probability, set from data; an SLA when formally agreed
Student syndrome	Starting late because the estimate contains safety
TameFlow	Steve Tendon's application of the Theory of Constraints to knowledge work
Throughput (T)	In this guide: the money the system generates through delivered work (Goldratt, Anderson)
Transaction cost	The fixed effort per batch: to release, test, review or approve
Utilisation	The share of available capacity in use
Weibull curve	A standard family of lopsided curves often used to describe Lead Times
WIP (work in progress)	Everything committed and not yet ready for delivery, including work that is waiting
WIP limit	A cap on how much work may be in progress at once
Work item age	How long an item still in progress has been open
WSJF	Weighted shortest job first: cost of delay ÷ duration
Zombie project	A project starved of resources but never formally stopped

Team Atlas: the whole case in one place

The guide uses Team Atlas as a running example. This section collects every Atlas fact from the chapters, in the order a manager would look at them. All figures are the same as in the chapters; illustrative figures are marked.

The team in numbers

WHAT	ATLAS	CHAPTER
Items delivered (ready for delivery) per week, 12 weeks	3, 6, 4, 0, 5, 7, 2, 5, 4, 6, 3, 8	Intro
Total delivered	**53 items**, about **4.4 a week**	Intro
Items committed per week	4, 4, 4, 5, 6, 1, 3, 8, 6, 6, 4, 4: **55 started**	5
Work in progress on an average day	about **6.2 items**	1, 9
Lead Time: most common value	**3 days**	1, 6
Lead Time: median / 70th / 85th / 95th percentile	**9 / 13 / 16 / 23 days**	6, 9, 11
Average Lead Time	**9.9 days**	1
Items left in the current release	**40**: about 6 in progress, 34 in the backlog	3, 11
Margin from the release once live *(illustrative)*	**€15,000 a week**	2, 3
Cost of running the team *(illustrative)*	**€8,000 a week**	3

What the numbers say

1. The flow is consistent with itself. Little's Law: 6.2 items in progress ÷ 4.4 a week ≈ 1.4 weeks, or about **9.9 days**. Atlas's actual average Lead Time was also 9.9 days (Chapter 1).

2. The "typical" item is a trap. The most common Lead Time is 3 days, yet about **75%** of items took longer. The median is 9 days (Chapters 1 and 6).

3. The tail is moderate, and it has a cause. Lead Times are lopsided with a moderate tail: median 9, 85th percentile 16, 95th percentile 23. The longest items were mostly delayed by **dependencies on other teams**. That is where the manager should look first (Chapter 6).

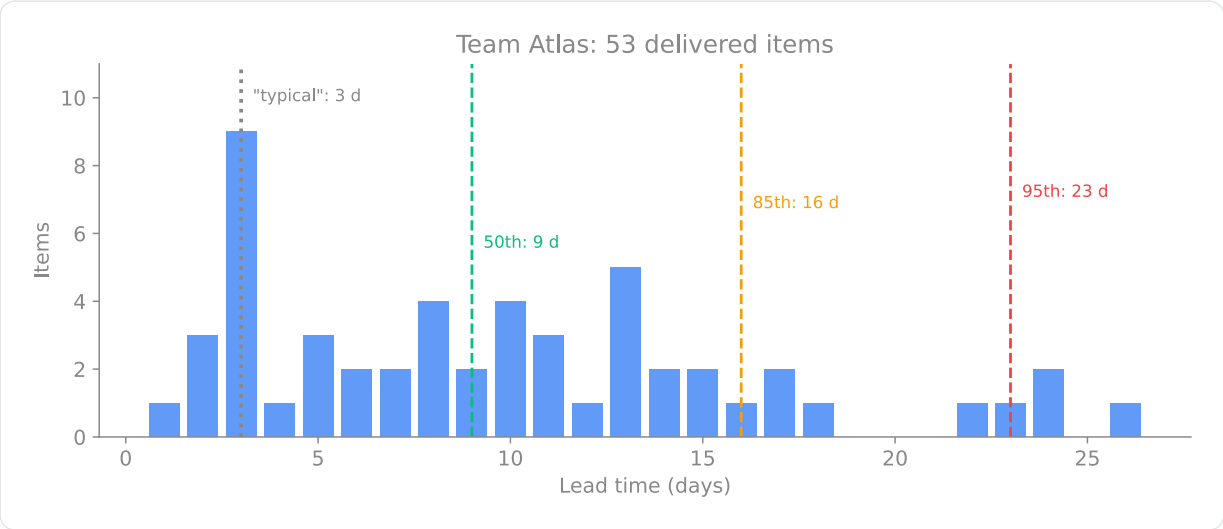


Figure 13 · Team Atlas Lead Times: median 9, 85th 16, 95th 23 days

Team Atlas, 53 delivered items: Lead Time distribution and percentiles.

4. Delay has a price. With an illustrative margin of €15,000 a week, every week of delay costs about **€15,000**. Starting two extra "quick" requests that push the finish out by a week costs about the same. "Can you just squeeze this in?" now has a price (Chapters 2 and 3).

5. Unfinished work is money waiting. The 40 remaining items are an investment that earns nothing until delivered. Idle time at Atlas's slowest step is worth far more than its share of the €8,000 weekly running cost (Chapter 3).

6. The team is loaded. In **7 of 12 weeks** Atlas started more than it finished: 55 started against 53 delivered. Weekly delivery swung from **0 to 8**. Nobody measures its utilisation, and nobody needs to: WIP and the cumulative flow diagram already show a loaded team (Chapter 5).

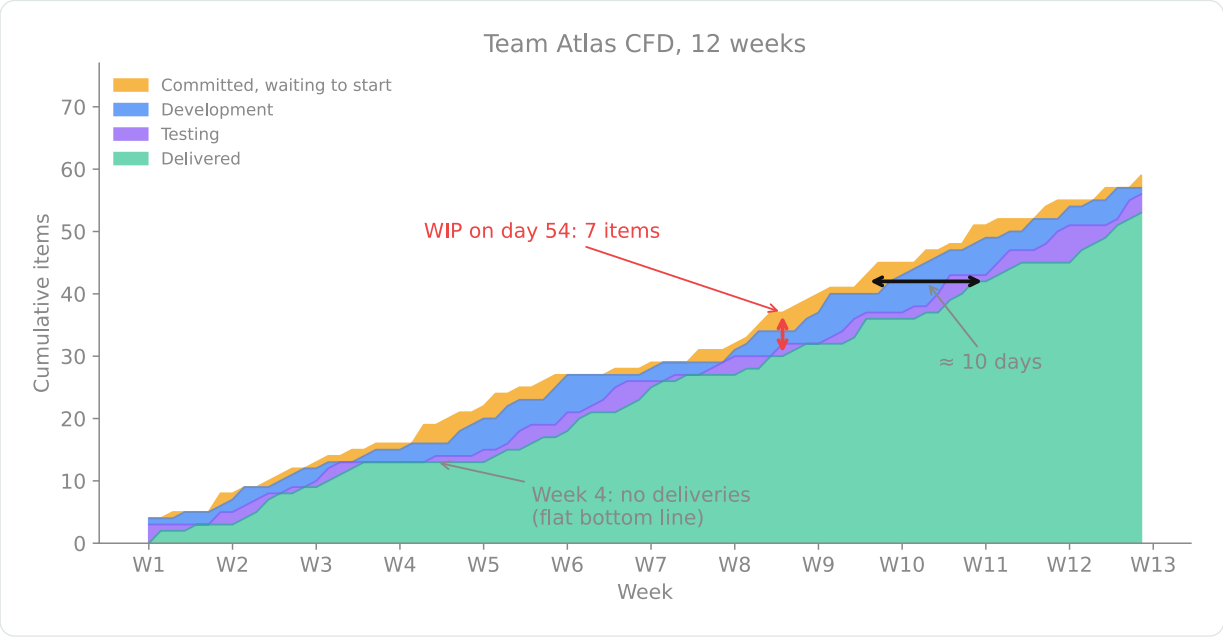


Figure 14 · Team Atlas cumulative flow diagram

Team Atlas, 12 weeks: cumulative flow diagram. The vertical gap is WIP, the horizontal gap is Lead Time.

7. One item needs attention now. At the end of week 12, six items are in progress. The oldest has been open **15 days**: just under the 85th percentile of 16 days (the SLE, Chapter 11), but above the 70th percentile of 13. With escalation at the 70th percentile, it would already be on someone's desk (Chapter 9).

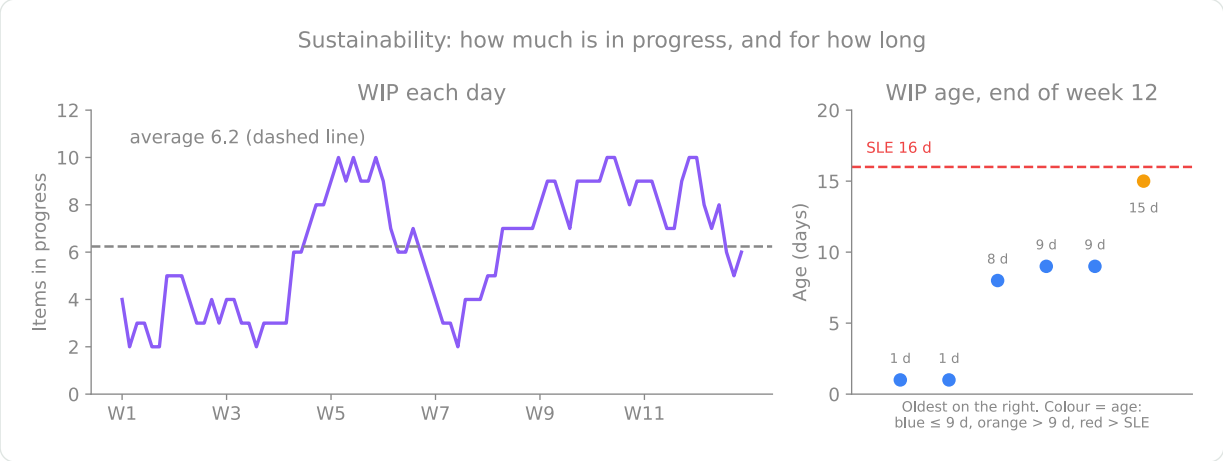


Figure 15 · Team Atlas WIP each day and age of items still in progress

Team Atlas: WIP each day (left) and age of the six items in progress at the end of week 12 (right).

8. A WIP limit would cost little. A light limit of about **12** items, twice the average, is invisible on a normal day and decisive in a bad week (Chapter 9).

9. The promise is a probability. Monte Carlo forecast for the 40 remaining items, from 12 weeks of history (Chapter 11):

CONFIDENCE	DONE WITHIN
50%	9 weeks
70%	10 weeks
85%	**11 weeks**
95%	12 weeks

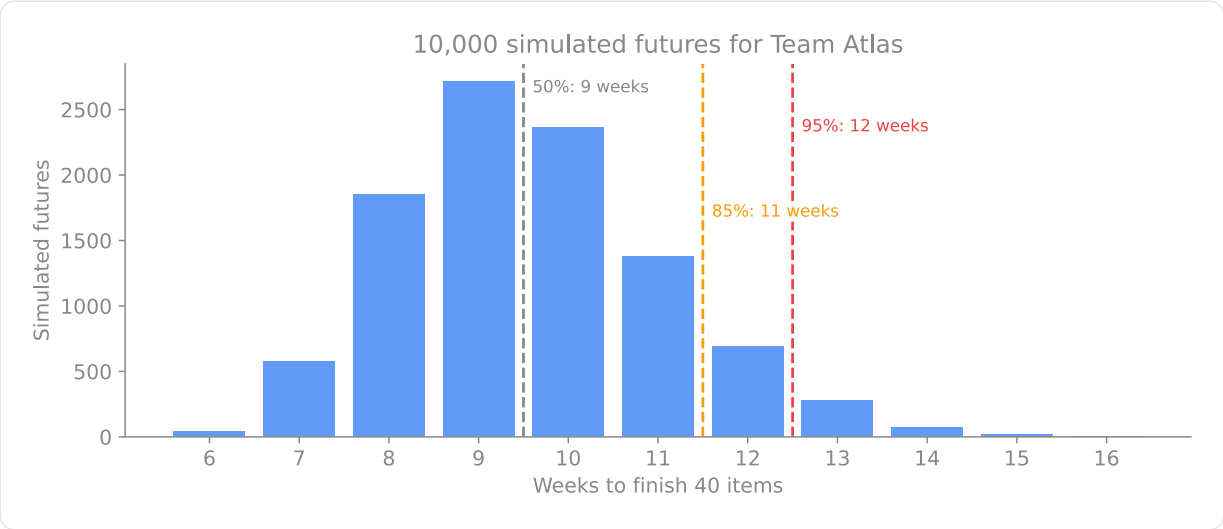


Figure 16 · Team Atlas Monte Carlo forecast for 40 remaining items

Team Atlas: Monte Carlo forecast for the 40 remaining items, 10,000 simulated runs from 12 weeks of delivery history.

The "average pace" answer, 9 weeks, is a coin toss. Re-run the forecast weekly.

10. A rhythm to reset. A fixed weekly review of WIP, oldest items and new requests, plus a separate replenishment meeting where new work is chosen by cost of delay (Chapter 12).

Atlas on one slide

What the manager of Atlas would bring to a monthly review (Chapter 13):

- about **6 items** in progress; the oldest at **15 days** against an 85th percentile of **16**;
- arrivals slightly ahead of deliveries: **55 started, 53 finished**;
- Lead Time median **9 days**, 85th percentile **16**;
- forecast for the release: **11 weeks** at 85% confidence;
- once a cost of delay is agreed, a **weekly price on the waiting**.

What the manager of Atlas should do on Monday

1. Look at the oldest item (15 days) and ask what it's waiting for.
2. Map the dependencies on other teams behind the longest items.
3. Set a light WIP limit of about 12 and stop starting more than the team finishes.
4. Agree a cost of delay for the release with the business.
5. Give the customer the 85% date, not the average one, and update it every week.

Sources

- Donald G. Reinertsen, *The Principles of Product Development Flow* (2009).
- Eliyahu M. Goldratt and Jeff Cox, *The Goal*.
- Eliyahu M. Goldratt, *It's Not Luck*.
- Eliyahu M. Goldratt, *Critical Chain*.
- David J. Anderson, *Agile Management for Software Engineering* (2003).
- David J. Anderson and Teodora Bozheva, *Kanban Maturity Model*, 2nd ed. (2020).
- Daniel S. Vacanti, *Actionable Agile Metrics for Predictability*, vols I and II.
- Daniel S. Vacanti, *Corporate Kanban* training material (2012).
- Troy Magennis, *The Economic Impact of Software Development Process Choice*; *The Economic Impact of High System Utilization*; *Understanding Optimal Batch Size*; *Understanding Cycle Time Distribution Shapes*; *Introduction to Monte Carlo Forecasting*.
- Alexei Zheglov, *Time in a Kanban System* (Lean Kanban Russia, 2015).
- Alexei Zheglov, blog *Connected Knowledge* (connected-knowledge.com), 2011–2019.
- Steve Tendon, *TameFlow Chronicles 2011–2015*.
- Steve Tendon, *Standing on Bits*.