CHAPTER 1

Concurrency, Parallelism, and Asynchrony Are Different

Three words. Three different questions. One chapter to make the distinction permanent.

This chapter is intentionally slower and more explicit than the chapters that follow. The reason is simple: nearly every later idea in this volume depends on separating concurrency, parallelism, and asynchrony correctly. ThreadPool behaviour, async I/O, bounded concurrency, queue growth, cancellation, backpressure, retries, and distributed failure all become harder to reason about when these three words are treated as synonyms.

The distinctions are not academic vocabulary. They describe different properties of a running system. Concurrency is about overlapping progress. Parallelism is about simultaneous execution. Asynchrony is about representing waiting without forcing the caller to remain synchronously occupied until completion.

THE CHAPTER CONTRACT

By the end of this chapter, you should be able to look at a C# method, a timeline, or a production trace and ask three separate questions: Are multiple activities in progress? Are multiple instructions executing simultaneously? Is waiting represented asynchronously? The answers can be different.

FIGURE 1.1 - THE THREE QUESTIONS

CONCEPT	QUESTION	WHAT IT DOES NOT IMPLY
Concurrency	Can activities overlap in progress?	Not necessarily simultaneous execution.
Parallelism	Can work execute at the same instant?	Not necessarily asynchronous waiting.
Asynchrony	Can incomplete work be represented without holding the caller?	Not necessarily another thread or parallel work.

CHAPTER 1

Start With an Everyday Analogy

Imagine one cook preparing two meals. The cook chops vegetables for Meal A, places them in the oven, then prepares Meal B while A is baking. The two meals overlap in progress. That is concurrency. Only one cook is actively manipulating food at any instant, so there is no parallel human execution.

Now add a second cook. If Cook 1 works on Meal A while Cook 2 works on Meal B at the same instant, the kitchen has parallelism. The two meals are also concurrent because their progress overlaps.

The oven introduces a third idea. After placing Meal A in the oven, Cook 1 does not need to stare at the tray until the timer expires. The cooking operation continues externally. The cook can do something else and return when the timer signals completion. That is the intuition behind asynchronous waiting.

FIGURE 1.2 - ONE KITCHEN, THREE DIFFERENT PROPERTIES

CONCURRENCY

One cook alternates between Meal A and Meal B.

Overlapping progress.

PARALLELISM

Two cooks work at the same instant.

Simultaneous execution.

ASYNCHRONY

The oven continues while the cook is free to do other work.

Waiting does not occupy the caller.

WHY THE ANALOGY MATTERS

The same operation can be concurrent without parallelism, parallel without asynchronous waiting, asynchronous without CPU parallelism, or all three at once. The properties are orthogonal.

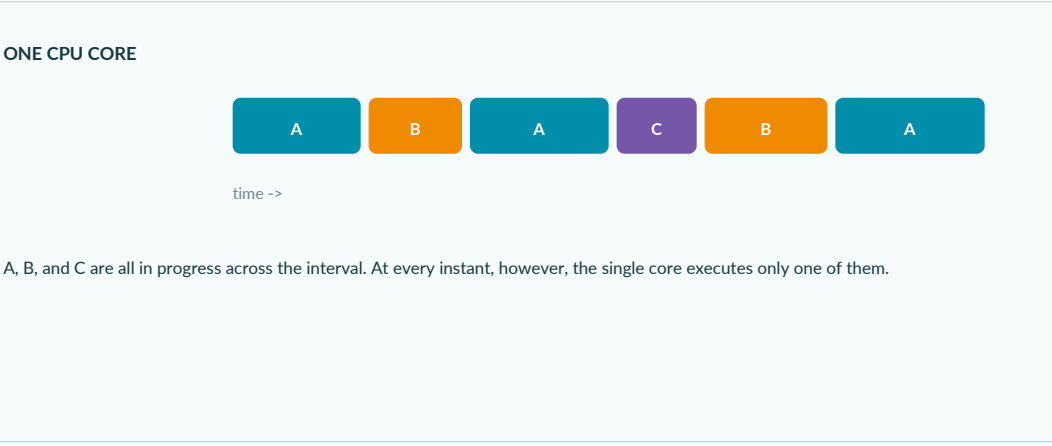
CHAPTER 1

Concurrency Does Not Require Two Cores

A single CPU core can execute only one instruction stream at an instant. Yet an operating system can maintain many concurrent activities by interleaving them. Thread A runs for a while, is preempted or waits, Thread B runs, and later Thread A resumes. Both activities exist during the same wider time interval.

This is why concurrency is fundamentally about the structure of work over time, not about the amount of hardware. A laptop with one core could still run a text editor, a music player, and a compiler concurrently by repeatedly switching which runnable thread owns the processor.

FIGURE 1.3 - CONCURRENCY ON ONE CORE



IMPORTANT CORRECTNESS CONSEQUENCE

Race conditions can exist on a single core. If operations interleave reads and writes in an unsafe order, the result can depend on timing even though two instructions were never executed simultaneously.

CHAPTER 1

A Race Can Happen Without Parallel Execution

Consider two logical operations that both increment the same counter. The source code looks harmless: read the current value, add one, write the result. But the increment is a compound operation. If the two operations interleave between the read and write, one update can be lost.

SOURCE

```
int counter = 10;

// Logical operation A
counter++;

// Logical operation B
counter++;
```

FIGURE 1.4 - AN INTERLEAVING THAT LOSES AN UPDATE

STEP	OPERATION A	OPERATION B	COUNTER
1	read 10		10
2		read 10	10
3	compute 11		10
4		compute 11	10
5	write 11		11
6		write 11	11

EXPECTED VS OBSERVED

Two increments suggest 12. The interleaving produces 11 because both operations read the same old value. This is a concurrency bug. Two cores can make the window easier to hit, but two cores are not required for the bug to exist.

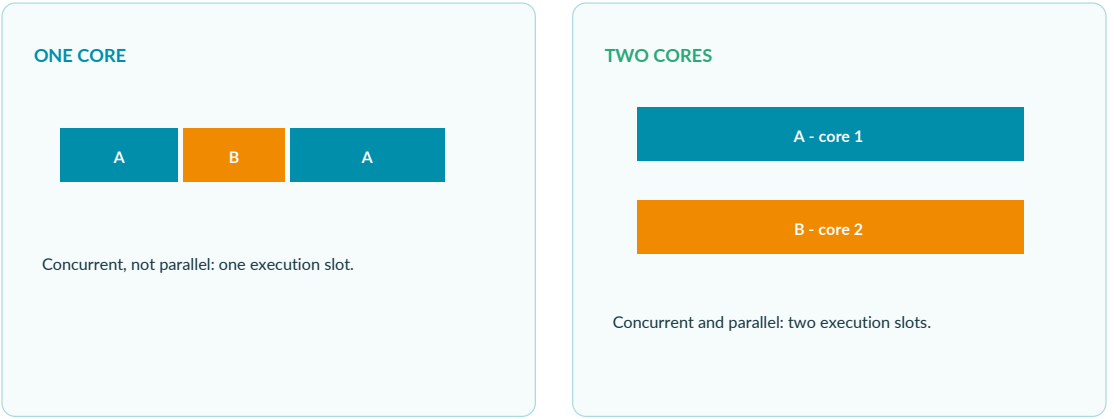
CHAPTER 1

Parallelism Requires Simultaneous Execution Capacity

Parallelism exists when multiple instruction streams execute at the same instant. Two CPU cores can run two threads simultaneously. A vector unit can apply one instruction to multiple data lanes. A GPU can execute many work items in parallel. In this book we will usually mean CPU-thread parallelism unless another mechanism is named.

Parallelism is useful primarily when there is independent CPU work to execute. If the workload spends most of its time waiting for a database, adding CPU cores does not make the database respond faster. If every worker contends on one lock, extra cores can amplify contention instead of improving throughput.

FIGURE 1.5 - INTERLEAVING VS SIMULTANEOUS EXECUTION



CAPACITY RULE

Parallelism consumes real hardware resources. Useful parallelism ends where another bottleneck begins: cores, cache, memory bandwidth, synchronization, I/O, or downstream capacity.

CHAPTER 1

Example: Parallelism Helps a CPU-Bound Calculation

Suppose an application must compute a CPU-intensive transformation for 8 independent images. Each image requires roughly one second of processor time. On a machine with enough available cores, processing several images simultaneously can reduce wall-clock completion time because the work is truly CPU-bound and independent.

SIMPLIFIED CPU-BOUND PARALLELISM

```
var results = images
    .AsParallel()
    .Select(ProcessImage)
    .ToArray();
```

FIGURE 1.6 - EIGHT SECONDS OF CPU WORK CAN BE PACKED DIFFERENTLY

EXECUTION SHAPE	APPROXIMATE WALL TIME	CPU TIME CONSUMED
1 worker, 8 images	~8 s	~8 CPU-s
4 workers, enough cores	~2 s	~8 CPU-s
16 workers on 4 cores	not ~0.5 s	still ~8 CPU-s + overhead

Parallelism changes how CPU time is scheduled across cores; it does not remove the work. If four cores are already saturated, creating sixteen compute workers does not create sixteen cores. It usually adds queueing, context switching, cache pressure, and coordination.

WHAT TO MEASURE

For CPU-bound parallel work, measure throughput, wall time, total CPU, runnable-thread pressure, contention, and memory traffic. Do not measure only the number of Tasks.

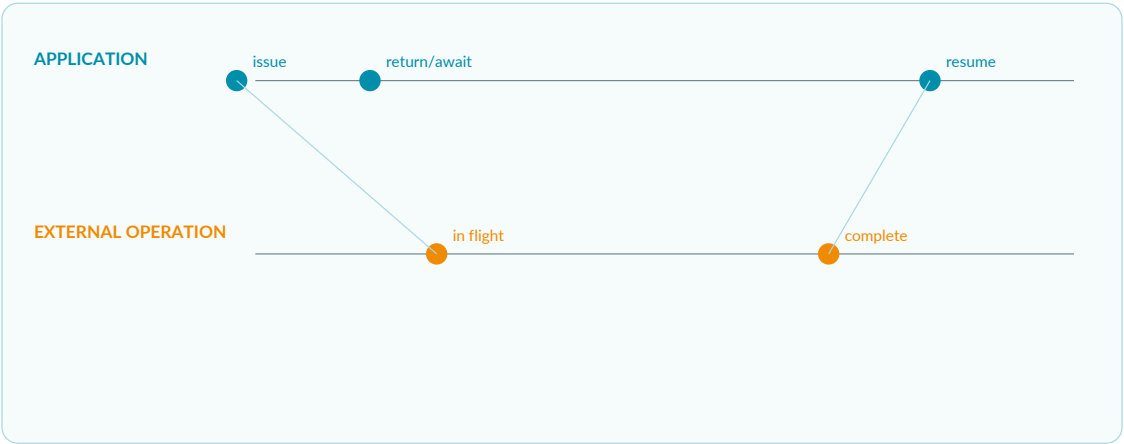
CHAPTER 1

Asynchrony Is Primarily About Waiting

Asynchrony answers a different question: can the program represent an operation that is not finished yet without requiring the caller to remain synchronously occupied until completion? This matters most when the delay is outside the current CPU instruction stream.

Examples include a network receive, a database response, a timer, disk I/O, a broker message, or a remote API call. The application cannot execute instructions to force these operations to complete immediately. It must wait for an external condition.

FIGURE 1.7 - THE WORK CAN CONTINUE OUTSIDE THE CALLER



ASYNC IS NOT THE SAME AS PARALLEL

While an HTTP request is waiting for the network, there may be no application instruction executing for that request. The operation is asynchronous, but that waiting interval is not CPU parallelism.

CHAPTER 1

Blocking and Asynchronous Waiting Have Different Costs

A blocking API keeps an execution context occupied while waiting. An asynchronous API can return an incomplete operation to the caller and allow the worker to be used for something else until completion becomes available. The external work may take exactly the same amount of time in both cases.

BLOCKING SHAPE

```
string Download()
{
    return httpClient
        .GetStringAsync(url)
        .GetAwaiter()
        .GetResult();   // caller blocks
}
```

ASYNC SHAPE

```
async Task<string> DownloadAsync()
{
    return await httpClient
        .GetStringAsync(url);
}
```

FIGURE 1.8 - SAME REMOTE LATENCY, DIFFERENT WORKER OCCUPANCY

PHASE	BLOCKING CALL	ASYNC CALL
request sent	worker occupied	worker executes
network wait	worker unavailable	worker can return to pool
response arrives	blocked worker resumes	continuation becomes runnable
remote duration	unchanged	unchanged

THE SCALABILITY GAIN

The benefit is not that the network became faster. The benefit is that waiting can stop consuming one application worker for the entire remote latency.

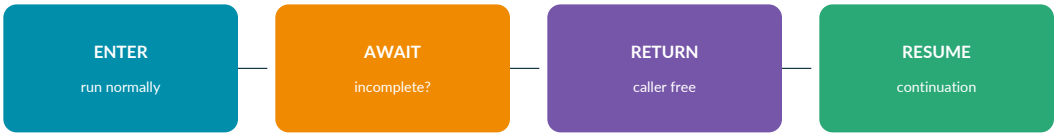
CHAPTER 1

What async/await Actually Changes

The C# compiler transforms an async method into a state machine. The method can run normally until it reaches an await. If the awaited operation is already complete, execution may continue immediately. If it is incomplete, the method records what is needed to resume later and returns control to its caller.

When the awaited operation completes, a continuation becomes eligible to run. That continuation may execute on the same thread or a different thread depending on the environment, captured context, scheduler, and completion timing. Thread switching is possible, but it is not the definition of await.

FIGURE 1.9 - ONE SOURCE METHOD, MULTIPLE EXECUTION INTERVALS



SOURCE LOOKS SEQUENTIAL; EXECUTION CAN PAUSE

```
async Task<int> ReadCountAsync()
{
    var text = await File.ReadAllTextAsync(path);
    return int.Parse(text);
}
```

NO MAGIC THREAD

The state machine explains why source code can look sequential while execution is split across time. It does not create a dedicated thread that sleeps inside the method.

CHAPTER 1

An Await Does Not Always Suspend

An awaited operation can already be complete when await examines it. A cache hit, an already-buffered result, or a completed Task can allow the async method to continue synchronously on the current execution path. This is one reason it is incorrect to describe await as 'always yielding the thread'.

FAST PATH + ASYNCHRONOUS SLOW PATH

```
async Task<string> GetNameAsync(int id)
{
    if (cache.TryGetValue(id, out var value))
        return value;        // may complete synchronously

    return await LoadFromDatabaseAsync(id);
}
```

FIGURE 1.10 - THE SAME METHOD CAN HAVE TWO RUNTIME SHAPES

CASE	AWAITED WORK	EXECUTION SHAPE
Cache hit	already available	method can complete immediately
Database miss	incomplete remote I/O	method can suspend and resume later

This matters for both performance and reasoning. An async method is not inherently expensive, but its completion path can vary. Code that assumes every await creates a scheduling break can be wrong; code that assumes no await ever creates one can also be wrong.

BETTER LANGUAGE

Say: 'await observes an awaitable. If it is incomplete, the method can suspend and arrange a continuation.' That wording remains correct across synchronous and asynchronous completion paths.

CHAPTER 1

Task, Thread, and Operation Are Different Things

A Thread is an execution context that the operating system can schedule. A Task is a .NET abstraction representing eventual completion. A logical operation is the business or technical activity we care about: process a request, save an order, read a file, publish a message. These three concepts can have many-to-many relationships.

FIGURE 1.11 - LOGICAL WORK CAN OUTNUMBER TASKS, AND TASKS CAN OUTNUMBER THREADS

CONCEPT	EXAMPLE	LIFETIME / OWNERSHIP
Logical operation	HTTP request	exists while the activity matters
Task	eventual HTTP response	exists while completion is represented
ThreadPool worker	executes continuations / CPU work	reused across many operations
OS thread	native schedulable context	managed by runtime + OS

A single request can execute on several workers over its lifetime. One worker can execute pieces of thousands of requests over time. Thousands of incomplete Tasks can exist while only a small number of workers are actively executing application code.

DIAGNOSTIC RULE

When a server slows down, 'there are many Tasks' is not enough. Ask whether Tasks are waiting, ThreadPool work is queued, OS threads are runnable, threads are blocked, or downstream operations are still in flight.

CHAPTER 1

Task.Run Does Not Turn CPU Work Into I/O

Task.Run schedules a delegate to execute, typically on the ThreadPool. It is useful when CPU work should be moved away from the current context, especially in UI applications. But the CPU work remains CPU work. Some worker must execute every instruction.

CPU WORK MOVED TO A THREADPOOL WORKER

```
await Task.Run(() =>
{
    // 500 ms of real computation
    ComputeLargeMatrix();
});
```

FIGURE 1.12 - SCHEDULING ELSEWHERE DOES NOT REMOVE THE CPU COST

The caller can remain responsive, but the process still spends the same compute instructions somewhere. On a saturated server, adding Task.Run can worsen scheduling pressure.

SERVER WARNING

Task.Run is not a generic scalability tool for ASP.NET Core. CPU-bound work still consumes ThreadPool capacity and CPU. Use it because the execution model requires it, not because the method should 'be async'.

CHAPTER 1

The Three Properties Can Combine in Different Ways

The cleanest way to make the distinction permanent is to examine combinations. Concurrency, parallelism, and asynchrony are not mutually exclusive, and none automatically implies the others.

SCENARIO	CONCURRENT?	PARALLEL?	ASYNC?
Two threads interleaved on one core	Yes	No	Not necessarily
Two CPU workers on two cores	Yes	Yes	No requirement
One HTTP request awaiting network I/O	May coexist with others	No CPU parallelism required	Yes
Many HTTP requests awaiting sockets	Yes	Not necessarily	Yes
Parallel image processing with Task.Run	Yes	Yes if cores available	Task API, but CPU work
One synchronous blocking file read	No extra logical concurrency required	No	No

THE MOST IMPORTANT READING HABIT

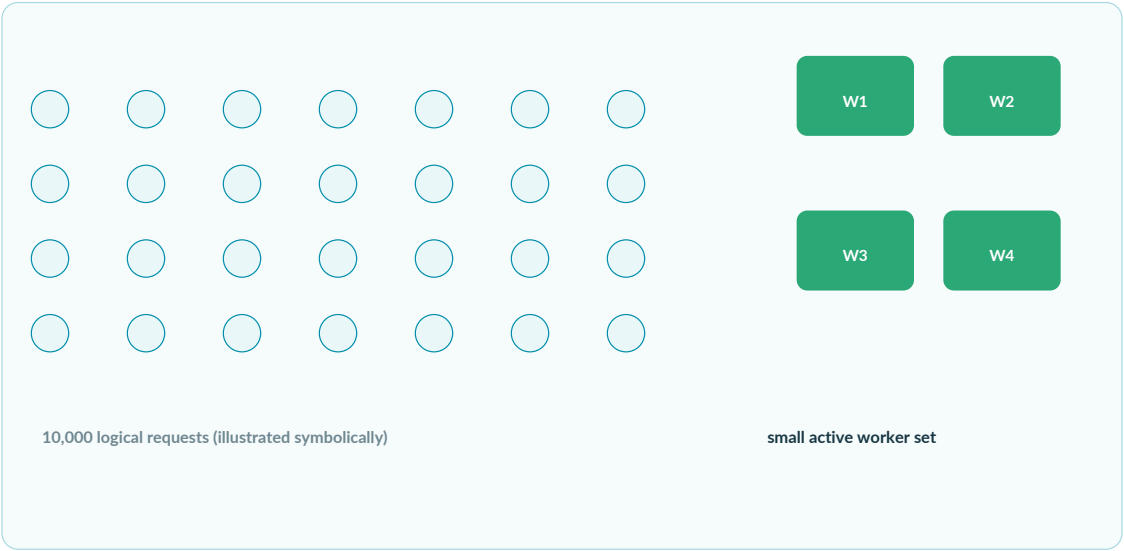
Do not classify a program by keywords. `async`, `Task`, `Parallel`, and `Thread` are clues about programming mechanisms. The actual properties come from runtime behaviour: overlap, simultaneous execution, and how waiting is represented.

CHAPTER 1

Example: A Web Server Handling 10,000 Requests

Suppose a server has 8 CPU cores and 10,000 active HTTP requests. It does not need 10,000 threads executing simultaneously. Most requests may be waiting for network input, a database, a remote service, or a timer. The logical request population can therefore be much larger than the worker population.

FIGURE 1.13 - MANY CONCURRENT REQUESTS, FEW ACTIVE WORKERS



The server is highly concurrent because many requests exist and overlap in progress. It may be parallel because several workers can execute on different cores. It is asynchronous when requests can wait for I/O without monopolizing a worker for the entire wait.

SCALABILITY IS QUEUE PLACEMENT

The engineering question becomes: where do the other requests wait, how much memory do they retain, and what resource eventually limits progress? Chapter 8 and Chapter 12 will make those queues explicit.

CHAPTER 1

Example: Desktop UI - Asynchrony Without Free CPU

A desktop UI has a special constraint: one UI thread commonly owns the visual state. If that thread performs a long synchronous operation, the window stops responding. Asynchronous I/O helps because the UI thread can return to its message loop while the external operation is in progress.

UI ASYNC EXAMPLE

```
private async void LoadButton_Click(...)
{
    status.Text = "Loading...";
    var data = await client.GetStringAsync(url);
    output.Text = data;
    status.Text = "Done";
}
```

FIGURE 1.14 - UI RESPONSIVENESS COMES FROM RELEASING THE UI THREAD

INTERVAL	UI THREAD	NETWORK
before await	starts request	request sent
while waiting	returns to message loop	in flight
completion	runs continuation	response available

Now replace the network call with a 5-second CPU calculation. Awaiting does not make that computation disappear. The UI can use Task.Run to move CPU work to a pool worker, preserving responsiveness, but a CPU still executes the calculation.

TWO DIFFERENT REASONS FOR TWO DIFFERENT TECHNIQUES

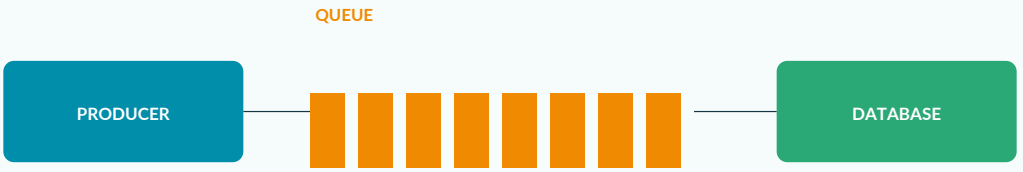
I/O asynchrony avoids blocking the UI thread while an external operation progresses. Task.Run can relocate CPU work away from the UI thread. Both improve responsiveness, but by different mechanisms.

CHAPTER 1

Example: Producer and Consumer - Concurrency Without More Throughput

Imagine a producer reading messages and a consumer writing them to a database. Running both concurrently can improve pipeline utilization: the producer can prepare the next item while the consumer is busy. But if the database can persist only 1,000 items per second, producing 10,000 items per second does not increase end-to-end throughput.

FIGURE 1.15 - CONCURRENCY CAN MOVE THE BOTTLENECK INTO A QUEUE



Concurrency improved overlap, but the slow stage still controls sustained throughput. If the producer keeps outrunning the consumer, the queue grows. Memory usage and latency grow with it. Eventually the system needs bounded capacity, backpressure, rejection, or shedding.

CONCURRENCY IS A CAPACITY DECISION

More simultaneous work is useful only while it increases productive utilization. Beyond that point it becomes retained work. Retained work is memory, queueing delay, timeout risk, and operational pressure.

CPU-Bound and I/O-Bound Are Resource Classifications

The distinction between CPU-bound and I/O-bound work is more useful than the distinction between 'sync method' and 'async method'. CPU-bound work consumes processor execution to make progress. I/O-bound work spends significant time waiting for something external to the current instruction stream. QUESTION CPU-BOUND I/O-BOUND

Progress requires	processor instructions	external device / service / timer
Typical examples	compression, hashing, image processing	HTTP, database, file, broker
Useful scaling tool	measured parallelism	asynchronous waiting + concurrency control
Main risk	CPU saturation / contention	queue growth / connection pressure / timeout
More concurrency	can oversubscribe cores	can increase in-flight work without more workers
Task.Run	moves compute to pool	usually unnecessary for true async I/O

CLASSIFY BY WHAT CONSUMES TIME

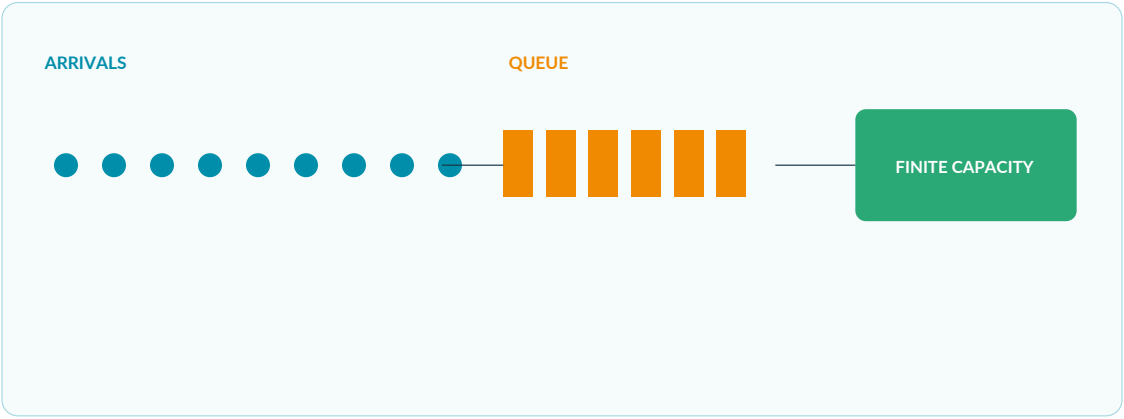
A method returning Task can still be CPU-bound. A synchronous method can be trivial and memory-only. Look at what the operation waits for and what resource is saturated.

CHAPTER 1

Concurrency Always Meets a Queue Somewhere

Every finite resource has a point where excess demand waits. A ThreadPool has queued work. A SemaphoreSlim has waiters. A database client has a connection pool. A socket has buffers. A broker has partitions and delivery queues. A remote service has its own internal queues. When arrival rate exceeds completion rate, pending work accumulates somewhere.

FIGURE 1.16 - ARRIVALS ABOVE CAPACITY BECOME QUEUED WORK



Concurrency is therefore not just a programming style. It determines how much work is admitted into the system at once. Increasing it can improve utilization when capacity is idle. Increasing it beyond the useful point converts demand into queue length.

LITTLE'S-LAW INTUITION

For a stable flow, more time in the system means more average work in flight at the same throughput. Longer waits therefore imply more retained requests, buffers, objects, and state.

CHAPTER 1

Bounded Concurrency Makes Scarcity Explicit

A concurrency limit says that only a certain number of operations may simultaneously consume a constrained resource. Additional demand must wait before entering, receive backpressure, or be rejected. This is not merely an optimization; it is a resource contract.

SIMPLIFIED CONCURRENCY GATE

```
using var gate = new SemaphoreSlim(8);

await gate.WaitAsync(ct);
try
{
    await CallDownstreamAsync(ct);
}
finally
{
    gate.Release();
}
```

FIGURE 1.17 - THE GATE CHOOSES WHERE EXCESS DEMAND WAITS

WITHOUT LIMIT	WITH LIMIT
Every caller starts downstream work	Only N calls enter downstream
Pressure appears in remote system	Pressure waits at a known local gate
Memory / sockets / connections can explode	Resource population is bounded
Failure emerges late	Admission policy becomes explicit

A NUMBER NEEDS A RESOURCE MODEL

Why 8? The answer should come from evidence: downstream capacity, CPU saturation, connection limits, memory, latency objectives, or controlled load tests. A magic constant is not a design argument.

CHAPTER 1

Backpressure Is the Other Half of Bounded Concurrency

If a component cannot accept unlimited work, upstream components need a signal or policy that prevents unbounded accumulation. Backpressure is the family of mechanisms that make downstream capacity visible upstream.

The mechanism can be blocking a producer, awaiting channel capacity, returning a rejection response, reducing fetch rate, limiting in-flight messages, or pausing consumption. What matters is that the system does not pretend capacity is infinite.

FIGURE 1.18 - WITHOUT BACKPRESSURE, THE QUEUE IS THE CONTROL MECHANISM

SYSTEM SHAPE	WHEN CONSUMER SLOWS	RESULT
Unbounded buffer	producer continues	memory + latency grow
Bounded channel	producer awaits capacity	pressure propagates upstream
Reject / shed	new work refused	latency protected at cost of admission
Adaptive source	producer reduces rate	flow follows observed capacity

PREVIEW OF CHAPTER 12

Later we will connect bounded concurrency, cancellation, channels, rate limits, and backpressure. Chapter 1 establishes the central truth: excess demand must wait, be rejected, or be dropped somewhere. Architecture chooses where.

CHAPTER 1

Cancellation Does Not Mean Undo

Asynchronous operations often live long enough that the caller may stop caring about the result. CancellationToken allows cooperative cancellation: a component can request that work stop, and code that observes the token can exit early.

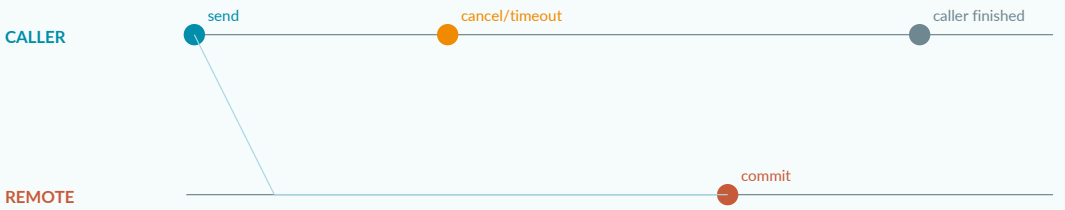
But cancellation is not rollback. If an operation has already written to a file, committed a database transaction, or sent a remote request, cancelling the local wait does not automatically reverse those effects.

LOCAL CANCELLATION BOUNDARY

```
using var cts = new CancellationTokenSource(
    TimeSpan.FromSeconds(2));

await client.SendAsync(request, cts.Token);
```

FIGURE 1.19 - THE CALLER CAN STOP WAITING BEFORE THE REMOTE SIDE STOPS WORKING



DISTRIBUTED PREVIEW

Timeout, cancellation, retry, and idempotency become one design problem after a process boundary. The caller's local completion state and the remote system's real state can diverge.

CHAPTER 1

A Timeout Describes Waiting, Not Reality

A timeout says: 'I will not wait beyond this deadline.' It does not necessarily say: 'the operation failed' or 'nothing happened'. The distinction is easy to ignore inside one process and impossible to ignore once another process or network is involved.

OBSERVATION	WHAT YOU KNOW	WHAT YOU DO NOT KNOW
Local timeout fired	caller stopped waiting at deadline	whether remote work completed later
Socket disconnected	connection ended	whether server committed before disconnect
Cancellation requested	stop was requested	whether every component observed it
Task cancelled	local Task has cancellation state	whether external effects were reversed

The key mental shift is that asynchronous and distributed systems separate the timeline of the caller from the timeline of the work being performed elsewhere. A local state transition can describe knowledge or intent without fully describing remote reality.

LANGUAGE DISCIPLINE

Prefer: 'the caller timed out' over 'the operation timed out' unless the external system's cancellation semantics are actually known. Precise language prevents false guarantees.

CHAPTER 1

Misconceptions to Remove Before Chapter 2

async means another thread	False. True asynchronous I/O may have no dedicated application thread while waiting.
await means thread switch	False. The awaited operation may already be complete; continuation placement varies.
Task means parallel execution	False. A Task can be incomplete, queued, waiting, or synchronously complete.
Task.Run makes I/O asynchronous	Usually false. It can move blocking work to a worker but does not change the underlying I/O mechanism.
more concurrency means more throughput	Only until a real capacity limit is reached.
one core means no race conditions	False. Unsafe interleavings are enough.
timeout means nothing happened	False across external boundaries.
parallelism is always faster	False when coordination, cache, memory, or another resource dominates.

CHAPTER 1

A Diagnostic Workflow for Real Systems

When a concurrent .NET system slows down, start by identifying the population and resource that are growing. Do not jump immediately to 'ThreadPool problem', 'async problem', or 'network problem'.

1. **How many logical operations exist?**
   requests, messages, jobs, timers, workflows
2. **How many are actively executing CPU work?**
   CPU utilization, runnable threads, hot stacks
3. **How many are waiting?**
   incomplete Tasks, blocked threads, I/O operations
4. **Where is work queued?**
   ThreadPool, channel, semaphore, connection pool, broker, remote service
5. **What resource is saturated?**
   cores, memory, connections, bandwidth, downstream throughput
6. **What is the admission policy?**
   unbounded, bounded, reject, shed, backpressure
7. **What does cancellation actually cancel?**
   local wait, queued work, remote request, committed effect

CHAPTER 1

Lab 1A - Make Concurrency Visible

Create two logical operations that repeatedly print a label and briefly yield or delay. First execute them one after the other. Then start both before awaiting completion. Observe that output can interleave even when the machine does not execute both at the same instant.

LAB 1A

```
async Task Work(string name)
{
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine($"{name} {i}");
        await Task.Delay(100);
    }
}

var a = Work("A");
var b = Work("B");
await Task.WhenAll(a, b);
```

OBSERVE

A and B overlap in time. Most of their lifetime is spent waiting for the timer. The program is concurrent and asynchronous. It does not require two CPU cores, and the timers are not consuming CPU for 100 ms each.

CHANGE ONE THING

Replace Task.Delay with a CPU loop that lasts roughly 100 ms. Run both operations directly, then through Task.Run. Compare CPU utilization, elapsed time, and thread behaviour. You have changed the mechanism from asynchronous waiting to CPU execution.

CHAPTER 1

Lab 1B - Make Parallelism Visible

Use a CPU-bound function and process several independent inputs. Measure the same workload with one worker and with controlled parallelism. The purpose is to see useful parallelism rise until hardware or another resource becomes the bottleneck.

LAB 1B

```
Parallel.ForEach(
    inputs,
    new ParallelOptions
    {
        MaxDegreeOfParallelism = 4
    },
    item => Compute(item));
```

RUN	MEASURE
degree = 1	wall time, CPU utilization
degree = 2	does wall time decrease?
degree = core count	does CPU approach saturation?
degree = 4 x core count	does throughput improve or only overhead?

EXPECTED LESSON

Parallelism is useful only while there is independent CPU work and unused execution capacity. Past the useful point, additional workers mostly redistribute waiting and overhead.

CHAPTER 1

Lab 1C - Compare Blocking and Asynchronous Waiting

Run many operations that wait using Task.Delay. Then compare with a controlled version that blocks workers using Thread.Sleep. The elapsed external delay can be similar, while worker occupancy is very different.

SAME APPROXIMATE DELAY, DIFFERENT EXECUTION COST

```
// asynchronous wait
await Task.Delay(500);

// blocking wait
Thread.Sleep(500);
```

FIGURE 1.20 - WHAT TO RECORD

SIGNAL	ASYNC WAIT	BLOCKING WAIT
incomplete logical operations	high	high
workers tied to waits	low	high
ThreadPool pressure	usually lower	can grow quickly
CPU during wait	low	low
scalability effect	more waits per worker population	worker starvation risk

DO NOT BENCHMARK THE SYNTAX

The lesson is execution shape, not that Task.Delay is a realistic database. In production, measure the real API and determine whether its implementation is completion-based or blocking underneath.

CHAPTER 1

Chapter 1 Checkpoint - The Mental Model to Keep

If only one page from this chapter remained, it should be this one. The terms below are the foundation for every concurrency and distribution mechanism that follows.

CONCURRENCY

Activities overlap in progress. One core is enough. The danger is uncontrolled interleaving.

PARALLELISM

Instructions execute simultaneously. It consumes real hardware capacity.

ASYNCHRONY

Incomplete work is represented without synchronously occupying the caller for the whole wait.

TASK

Represents completion. It is not a thread and not a promise of parallelism.

ASK THESE THREE QUESTIONS SEPARATELY

1. Are multiple activities in progress during overlapping time?
2. Are multiple instruction streams executing at the same instant?
3. Can incomplete work release the caller until completion?

If you can answer all three independently, the vocabulary is stable.

NEXT - CHAPTER 2

Shared Memory Means Shared Correctness. We now take concurrency's most important consequence seriously: multiple execution paths can observe and modify the same state, so correctness depends on ordering, visibility, and atomicity.