CLOUD DONE CORRECTLY

capacitas®

COST | SCALABILITY | PERFORMANCE

WHITEPAPER

Creating Cost Efficient Software

Index

Audience

This white paper is aimed at those responsible for architecting and writing software that will run in the cloud.

Introduction

"The cloud is just someone else's computer, but it is a rental computer."

Let's get straight to the point; the code you write for software deployed in the cloud directly impacts the monthly cloud bill. There was a much looser relationship in the past when hardware was on-premises. The hardware had been purchased years ago, and unless your software changes meant there was a need to buy more servers, nobody cared. In fact, the infrastructure manager probably liked you to use their hardware as it justified their decision to buy it all. Now, however, if a resource is over or underused, it directly relates to the monthly cloud bill.

The wildest days of the cloud have gone, with companies rushing to the cloud as the next big thing. The giddiness of the new technology has been replaced with the dark reality of expensive monthly bills with too many unpredictable spikes in cost. There are many news stories of an exodus back to on-premises data centres, but when you read the details, they are about specific workloads and business cases.

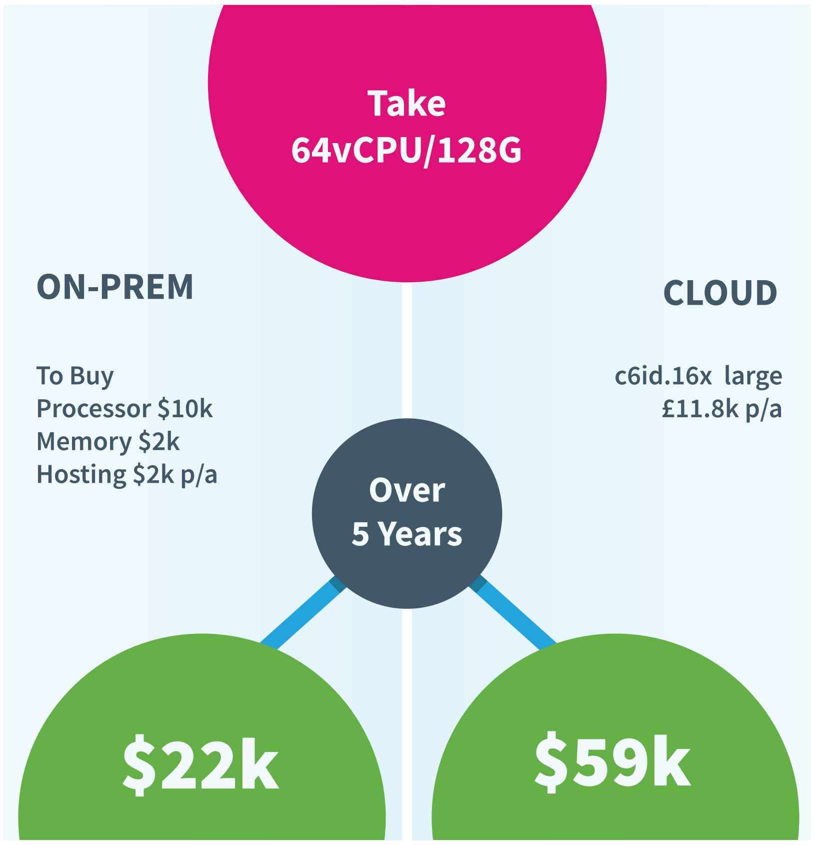
The reality is that the cloud is here to stay, and it is now about making sure you design, code and operate your applications to maximise the benefits of the cloud to produce cost-efficient services.

The cloud is naturally cheaper, right?

Here is a quick calculation: Take the c6id family of AWS instances that use the Intel(R) Xeon(R) Platinum 8375C CPU @ 2.90GHz. This is a 64vCPU processor, so that would map to the c6id.16xlarge instance that costs $11.8k per year if you commit to three years. Over 5 years, the TCO would be $59K. For the on-premises costs, the processor costs $8,000 to $10,000 when new, and let's put that in a server with 128G of memory that will cost about $12K and then your cost to rack and stack that server per year is about $1K per year. Assume after 5 years the server is worthless, then you are $17K down. Now, there are other costs like finance and support costs, but let's consider the TCO, which could be around $25K. Of course, if you used that compute power for just one day out of the 5 years, then AWS would have cost you $77 compared to $25,000 for running on-premises.

So, depending on your hosting and support costs, you could probably buy and run a server cheaper than committing to buying the same level of compute through a primary cloud provider. Hence, the **challenge is to make the move to the cloud worth it.** What this illustrates is you must have processes in place to manage capacity and develop your software to maximise what the cloud offers.

Now, it is not only the responsibility of the software engineer to efficiently control costs. Just like regular software design, the business must accurately provide inputs, such as the expected workload into the system. By the way, the workload is more than just the number of users or transactions per second; it also has a shape, particularly how the demand varies over the day, week, year, etc. The shape of the demand curve makes a big difference in the decision you need to make as a software engineer. Finally, let's not forget the response time requirements we give the users.



I am sure you have heard the phrase the cloud is just someone else's computer. To me, that is the challenge and opportunity of the cloud. Think of it as a rental car; it is just someone else's car, and the world of possibilities opens up. You could probably pay less over five years to own your own car versus having a rental for all that time.
But you don't have to rent that car for five years, you could have two cars when you need them, none when you don't. You could take the bus when you are not in a rush or take a plane when you are.

To write a cost-effective system, you need to remember your compute resource is a rental transport solution. You get a discount for committing long-term, and you can flex or even have fancy services. This is where you have the opportunity with the cloud to make it cost-efficient.

As software developers, it is not just about writing efficient code but also maximising the benefits of having rental computers.

This white paper will help you think about and plan for writing cost-efficient software. It will examine both the software architecture design, as well as look at the more detailed code level.
But first, let's start with the money.

Chapter 1
Every Line of Code Costs

Of course, everything costs. In 2023, AWS generated a total of $90.8 billion in sales, Google Cloud $33.7 billion and Microsoft $96.8 billion in sales.

In this chapter, I provide some basic information about some of the cloud services and their costs. It is important that you start to think about the choice of cloud service for your software design, as this starts to have a major impact on the cost.

"Price is what you pay. Value is what you get."

– Warren Buffett

Take, for example, storage of documents which in the past may have been held in the database. They may be stored more cheaply in object storage in cloud using object storage (S3, etc). However, you will see you need to consider all parts of the billing model to determine if that is the best choice.

Below are some general costs based on typical cloud service pricing (from providers like AWS, Azure, and Google Cloud). Please note that actual costs can vary depending on the provider, region, discount and length of commitment:

https://www.crn.com/news/cloud/2024/aws-vs-microsoft-vs-google-cloud-earnings-q4-2023-face-off?page=2

Item	Estimated Cost
4CPU Compute Optimized	$0.10 to $0.20 per hour
4CPU Memory Optimized	$0.15 to $0.25 per hour
Storage of 1GB on Fast Disk (SSD)	$0.10 to $0.20 per GB per month
Storage of 1GB of S3 (Object Storage)	$0.023 per GB per month (AWS S3 standard storage)
Egress Network Traffic (Per GB)	$0.08 to $0.12 per GB
Single Call to a Serverless Function (0.5 sec)	$0.0000002 per invocation (AWS Lambda)
Monthly Cost for a Publicly Available IP Address	$3 to $5 per month
Storing a 1GB Log File	$0.03 to $0.10 per GB per month (depending on retention period)
Storing a Custom Metric	$0.30 to $0.50 per metric per month

For more accurate costs, it's recommended you use a cloud provider's pricing calculator like AWS Pricing Calculator, Google Cloud Pricing Calculator, or Azure's Pricing tool.
Alas, the cost model for service providers is not purely one-dimensional, e.g. priced on data stored. The cost varies depending on the service level offered and other factors, such as the frequency and type of accesses to that data.

Take for example object storage like S3.

1. Storage costs

This is the base cost, which typically depends on the amount of data stored per GB. For example, AWS S3 charges around $0.023 per GB per month for standard storage, with lower rates for archival tiers like S3 Glacier or S3 Glacier Deep Archive.

2. Data retrieval and access

- Frequency of Access: If you're frequently accessing data, especially in large volumes, you'll incur additional charges. Different S3 storage classes are designed for different access patterns:
    - S3 Standard is optimised for frequent access, but if you choose S3 Infrequent Access (IA) or S3 Glacier, you pay lower storage fees but higher retrieval costs when you access the data.
- Data Access Types: Costs also depend on the type of operation.:
    - GET requests (retrieving data): Every time you read an object from S3, it counts as an API request. With millions or billions of small files, these costs can add up, as you're billed per 1,000 requests (about $0.005).
- PUT, COPY, POST, DELETE requests (modifying or uploading data):
    - Similarly, you incur charges every time you upload or modify data, even though these operations might be cheaper than GET requests (about $0.0004 per 1,000 requests).

3. Data transfer (egress costs)

Moving data out of S3 to the internet (egress traffic) is not free and can add a significant cost, especially if you're serving large files to external users. S3 has a data transfer pricing model where you are charged for each GB moved out of the storage bucket to other AWS services, external networks, or even other regions.

4. Lifecycle transitions

S3 offers the ability to automatically transition objects between storage classes (e.g., from S3 Standard to Glacier) based on usage patterns, but these transitions are not free. Lifecycle management can incur costs if there are frequent transitions.

5. Request and metadata overhead

Along with data retrieval, the storage cost also involves charges for list requests, metadata retrievals, and even checking the existence of objects in a bucket. Each interaction with the storage, even for simple metadata requests, counts toward the number of operations billed.

6. Multipart uploads and unused data

For large files, multipart uploads are used to upload data in parts. If a multipart upload fails to complete, the stored parts are charged as separate objects unless manually deleted.

When it comes to compute resources, remember that the cloud is someone else's computer, a rental and flexible one.

Cloud purchasing models

1. **Serverless (Lambda):** Pay only for exact execution time. Ideal for infrequent workloads or scaling on-demand with minimal management. You're charged per millisecond, which eliminates the need to maintain servers constantly.
   - Cost model: Pay per invocation, great for event-driven applications.
2. **EC2 On-Demand:** Instant access to compute capacity with no long-term commitment. Perfect for unpredictable workloads where flexibility is key.
   - Cost model: Pay-per-hour (or second), higher flexibility, but at a premium compared to reserved models.
3. **EC2 Reserved Instances:** For workloads you know will run continuously, 1 or 3-year reservations can lead to significant savings (up to 72%). You trade upfront commitment for lower long-term costs.
   - Cost model: Discounted hourly rates in exchange for term commitment.

EC2 Spot Instances: Get unused compute capacity at up to 90% off. Spot instances are great for fault-tolerant workloads or batch processing that can handle interruptions However, the trade-off is the uncertainty of availability.
- Cost model: Extremely low cost, but your instance can be terminated if AWS needs the capacity back.

If you design part of your software to be easily interrupted and restarted you could save big by using spot instances. Or if you get massive, short spikes in demand, maybe you need Lambda.

Cautionary Tale

Not understanding how services are costed can lead to unexpected spikes in the bill. In his article, software engineer Maciej Pocwierz tells of how an S3 bucket unexpectedly had a massive spike in cloud costs due to unexpected (and, in this case, unauthorised) requests to the S3 bucket.

Golden Rule

If you are not getting value from the service you are using, choose a different service. For example, imagine you have to store loads of user documents that are produced from survey taking; needed but rarely retrieved. Don't put these in a cloud database. Put them in object storage like S3. Typically, it is 10x cheaper as long as you are not frequently accessing them.

Chapter 2
Code to Match the Workload

"Assumptions are the termites of relationships."

– Henry Winkler, Actor

With the cloud, everything costs. But remember, we are thinking about these computers as rentals. There are many different service models you can choose from. The trick is to choose the right model for the workload. What do we mean by workload? This is the profile of the demand arriving at your system over time.

Systems experience various types of workload arrival patterns, each driven by user behaviour, time of day, and specific events. One common pattern is the **daily workload** cycle, where system activity fluctuates within a 24-hour period. Such workloads often peak at predictable times, such as just before and after lunch, when users engage in routine activities like email checks, data entries, or report generation. These systems might show a lull in activity during mealtimes or late at night. An example would be corporate systems or websites with regular business hours that see most of their traffic between 9 AM and 5 PM.

Another distinct pattern is a **spiky or bursty workload**, which features sudden, intense spikes in activity at specific intervals. These spikes are often associated with special events, such as ticket sales, product launches, or flash sales, where thousands or millions of users try to access the system simultaneously within a short time frame. These systems require robust scaling capabilities to handle the abrupt surges in traffic without overloading. Conversely, some systems may experience a constant or steady workload, where the load remains relatively uniform throughout the day. These systems are typically used for services that need to be accessible around the clock, such as public APIs or monitoring systems.

Seasonal workloads represent yet another category. These systems show pronounced activity peaks at certain times of the year. For instance, an online tax-filing system would see minimal usage for most of the year but experience a dramatic increase in traffic leading up to the tax submission deadline. Similarly, retail systems experience heightened activity during holiday seasons like Black Friday or Christmas. Understanding these patterns helps in designing systems that can scale dynamically to handle varying loads, ensuring optimal performance and user experience.

Even versus spiky workloads – putting them into context

Consider this example when it comes to even vs. spiky workloads, with prices based on AWS. Assume you need to process 1,123,200 requests per day. If the requests arrive evenly, you can benefit from purchasing an EC2 Reserved Instance. But if the workload is spiky, a serverless approach like AWS Lambda might be more cost-effective.

Example Calculation

You have an API request that takes 1 second and uses 1 CPU second along with 512MB of memory.

Compute Instance

For an even workload, that's 46,800 transactions per hour. Since each transaction takes 1 CPU second, that equals 46,800 CPU seconds per hour. A 16vCPU instance provides 57,600 CPU seconds per hour (16 * 3,600), which results in 80% utilisation — near the optimal performance threshold before degradation occurs. In this case, you've hit the sweet spot.
AWS offers a 16vCPU/32GB Graviton instance for $8.40/day as a Reserved Instance (with a one-year commitment).

Serverless Option

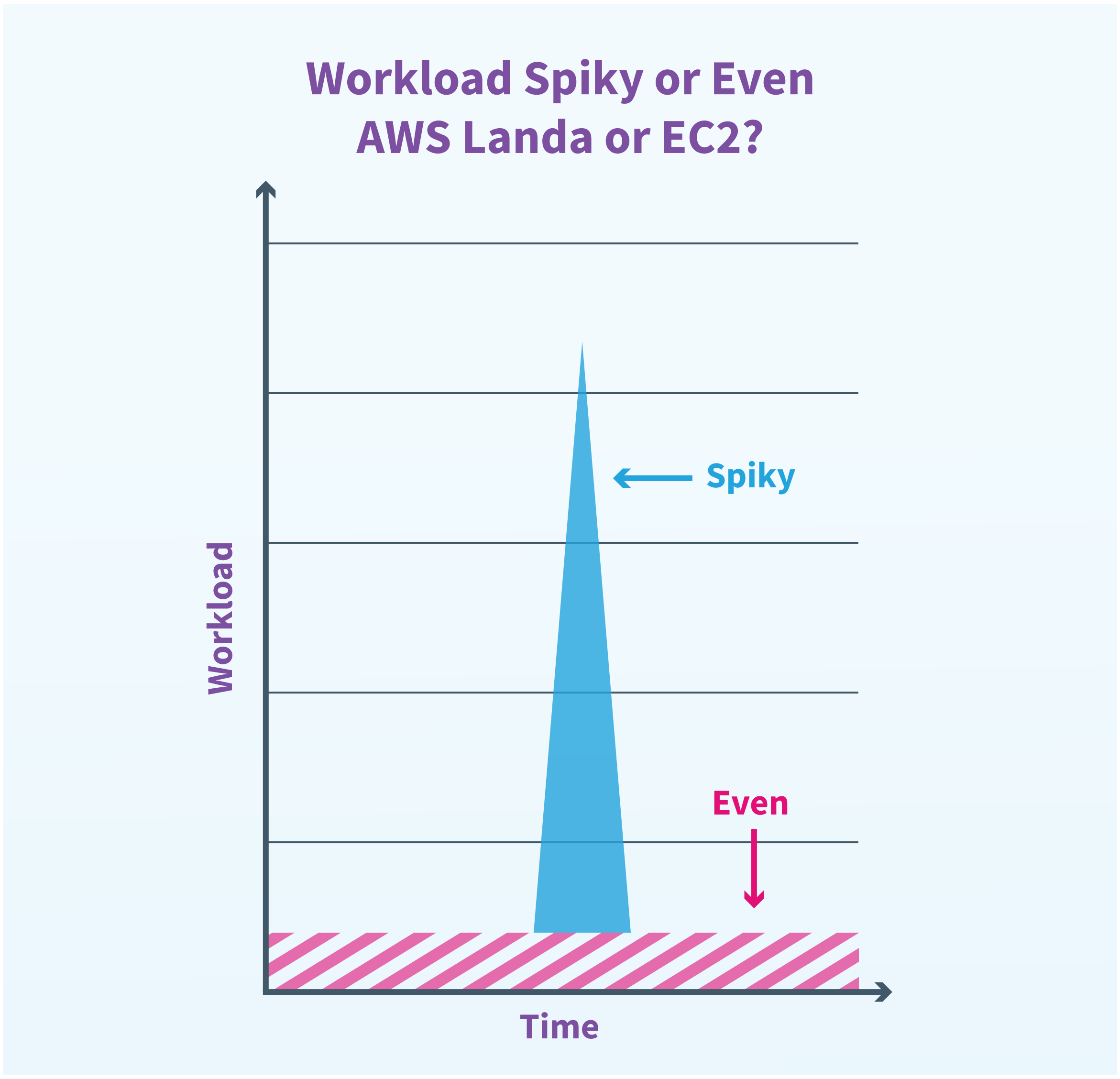
With AWS Lambda, the cost is based on memory usage and duration, plus a fee per transaction. A single 512MB, one-second request costs $0.0000067, and for 1,123,200 daily transactions, the total cost is $7.73, including a $0.20 fee per million requests.

Surprisingly, Lambda is competitively priced in this scenario, compared to an even workload running on a Reserved Instance.

When accounting for spiky workloads using On-Demand and Reserved Instances, I've estimated the use of 24 x 16vCPU boxes for simplicity. As expected, Lambda comes out as the winner for spiky workloads.

Arrival Pattern	Compute	Daily Cost
Even Arrival	1x 16vCPU Reserved Instance	$8.40
Even Arrival	Lambda @512MB	$7.73
Spike Arrival	Lambda @512MB	$7.73
Spike Arrival	24x 16vCPU On-Demand	$13.20
Spike Arrival	24x 16vCPU Reserved Instances	$201.60

This doesn't mean compute instances are obsolete, and you shouldn't always default to Lambda. The billing model is complex, and I haven't included storage or network costs. But it's interesting to see how costs change when the API becomes more memory-intensive with daily cost being $14.94 at 1024MB and $29.99 at 2048MB.



What am I designing to?

The above example shows the importance of mapping the right service to the workload. Understanding what services will be your biggest cost drivers is also important. Once you know these, you can focus on them to make sure you are picking the appropriate services and solutions to efficiently control your costs. Of course, the earlier you can identify the cost driver, the better you are at controlling them. Unfortunately, early in the project, there are often many unknowns and assumptions.

However, $\text{Cost Per Active User} = \dfrac{\text{Cloud Costs}}{\text{Number of Actice Users}}$

it is always worth doing a bit of napkin math when making architectural decisions. Of course, I am a number geek, so I will often use Excel (see next page). Remember, each project is different.

Network	Monthly	Unit Cost	Units	Total
Egress (Data out of the cloud)	2000	0.05	GB	100
Inter-Region (Geo - Regions)	1000	0.02	GB	20
Inter-Zone (Transfer across availability zones)	500	0.01	GB	5
Static Costs (IP costs, etc)				
Compute				
Serverless	10000000	0.000007	Invocations	70
On-Demand				
Compute Optimised	40	25.92	per CPU	1,037
Memory Optimised	40	38.88	per CPU	1,555
Reserved/Saving Plan (Typically 30% cheaper)				
Compute Optimised	80	18.144	per CPU	1,452
Memory Optimised	120	27.216	per CPU	3,266
Spot Instances (Typically 70% cheaper)				
Compute Optimised		7776	per CPU	0
Memory Optimised		11664	per CPU	0
Storage				
Object Storage	5000	0.005	per GB	25
Block Storage (SSD, HDD)	10000	0.1	per GB	1000
Really Really Fast Storage	5000	0.4	per GB	2000
File Storage/Backup	75000	0.2	per GB	15000
Access Costs (10% of Storage)				
Other				
Very project dependent...				
			Total PCM	25,529

As a rule of thumb, the ratio should be 50% compute, 30% storage, 10% network, and 10% other.

These calculations just need to be rough. This is about getting you into the mindset of understanding what drives your costs and where you may need to look to optimise. For example, you discover your storage costs are significant, so think: Are you using the most appropriate cloud service to store your data? Can you reduce the data size, etc.?

Cautionary Tale

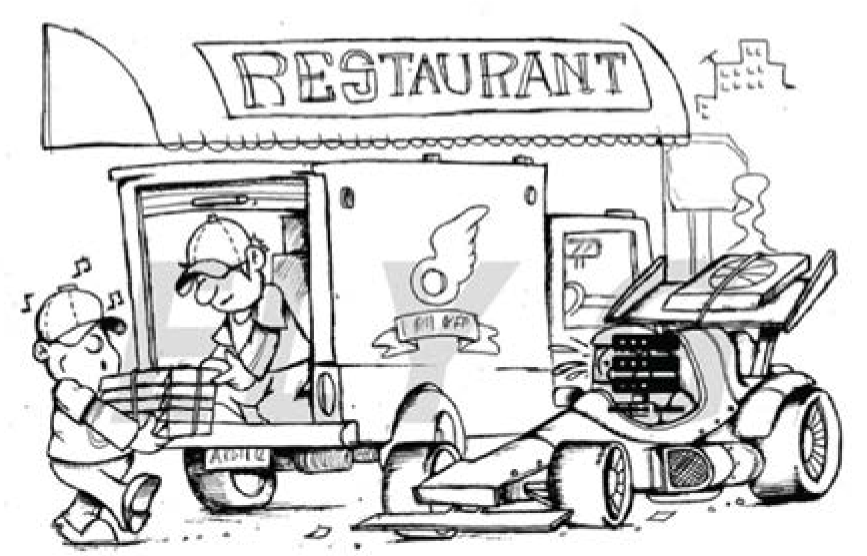
A customer redesigned their software to automatically scale compute resources with demand. However, their workload demand was pretty static during the working week and minimal at the weekend. They could have saved that effort if they had just written a script to turn off some computers at the weekend!

Golden Rule

Do rough cost calculations **early** in the project to help steer the architectural decisions. Don't get sidetracked by trying to make these overly accurate. This is all about identifying major cost drivers and making sure you are making appropriate cost-saving architectural decisions.

Chapter 3
Efficiency is Just Performance, Right?

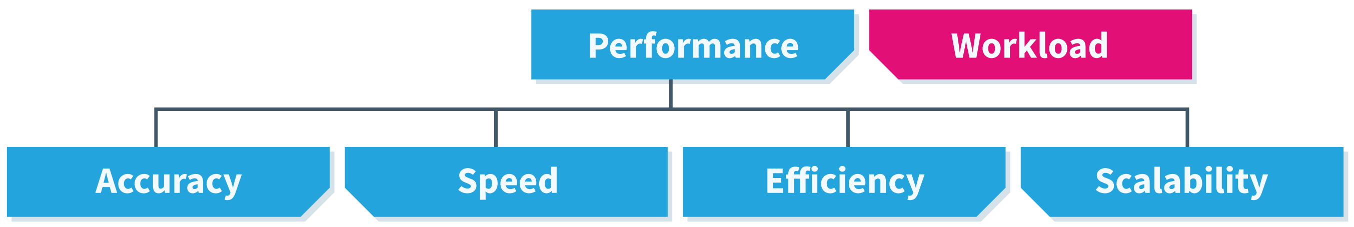
The early bird may get the worm, but the second mouse gets the cheese!



Although having the fastest car Bob the Uber Eat$ driver couldn't beat the van for the bulk Christmas party delivery.

Surely, performance and efficiency are the same?

Let's start at the beginning. Performance is a broad term, which many people often associate solely with speed (i.e., "Is it fast enough?"). However, there are more elements involved, and it's crucial to consider how the system reacts under different workload levels. For example, users might experience good response times when there are only a few users, but the system may become significantly slower when many people are using it simultaneously.



In principle, software performance refers to "how well the software runs" and consists of four core execution elements that you can either improve or sacrifice:

- Accuracy: The number of errors that occur while executing a task. A system that quickly returns HTTP 500 errors is not a high-performance system.

- Speed: How fast the work is done to complete a task. This can be observed in terms of response times or throughput. Response time refers to the time taken to execute a task, while throughput indicates the number of tasks completed per time unit.

- Efficiency: A measure of the resources used to complete a task. If one sorting algorithm takes 5 CPU seconds compared to another that takes 50 seconds, the first is more efficient. You must remember that efficiency in relation to cost has multiple dimensions, such as CPU, memory, and network usage, all of which have associated costs.

- Scalability: The ability of the system to handle an increased volume of workload.

Is more efficient code faster?

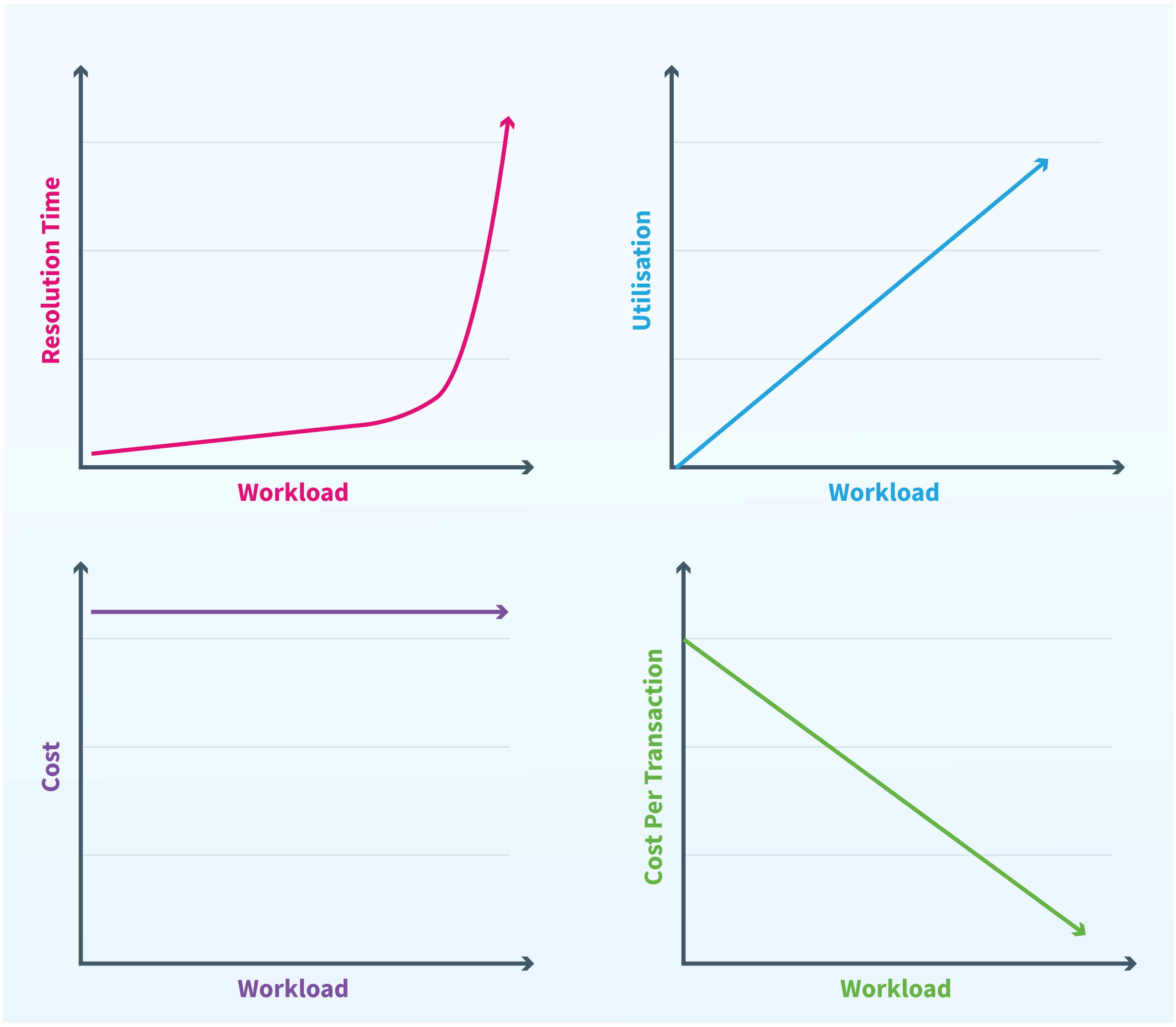
This is generally true. The fewer resources you use, the quicker the code will execute. However, this is not always the case — sometimes, we sacrifice efficiency to improve speed. For example, we might cache results to reduce response times, but this could negatively impact memory efficiency.

Next, you need to consider how to maximise your resources. There's little point in optimising code to use less CPU, for example, unless you can actually leverage the cloud and downsize the amount of CPU you use.

Before we continue, I need to provide a bit of background on the concept of the utilisation sweet spot.

Unlike storage space, where you can be at nearly 100% utilisation without degradation, CPU utilisation is more complex. The graphs below illustrate the relationship for a single server. In all four graphs, the x-axis is an increasing workload e.g. number of users or transactions per second.

The top left graph shows response time, and the key here is that at around 70-80%, response times will degrade probably to the point users will complain. At the 70-80% point this is where you have hit the sweet spot of the lowest cost per transaction. You have a static cost for the server as seen in the bottom left, but a decreasing cost per transaction as you increase workload see bottom right.



Cautionary Tale

Most major cloud providers offer compute resources in what they call “t-shirt sizes.” Typically, these sizes double the number of CPUs as you move to the next larger box. Therefore, efficiency gains often need to exceed 50% to justify a change in size. The good news is that if your software is currently inefficient, achieving these gains is likely possible.

Golden Rule

Aim to maximise the usage of all your compute resources and remember to do this as your workload increases and decreases throughout the day.

Chapter 4
When Do I Optimise?

"Any design, whether for a bridge or a building, must start with a solid foundation. If the foundations are not right, everything else will be wrong."

- Henry Petroski, Engineer

Not every line of code needs to be efficient. You are losing money if the software costs more to make efficient than it saves. However, as we know, if we don't start with firm foundations, we will never get it right. I have always liked Connie Smith's (author of Performance Engineering of Software Systems) example, which says it is much easier to build an energy-efficient house from the ground up rather than retrofit (I can relate to living in a house over 350 years old and is painful to heat).

However, there is also Donald Knuth's view from his book Computer Programming as an Art (1974) that "premature optimization is the root of all evil." Interestingly, the full quote from the book is more nuanced: "The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming."

So, who is right? When do you start writing efficient code? Well, I think both Donald and Connie are right. You just need an approach proportional to what you want to achieve. The key to successful optimisation is focusing on efficiency in the right places.

From the beginning, you must understand the big cost drivers and choose an appropriate initial architecture/design. Of course, this is one of those things that is easy to say but difficult to do as you are dealing with a lot of ambiguity at this stage. I recommend listing the known drivers and the assumptions you are making to aid clarity and allow stakeholders to correct the fundamentals.

Communicate the efficiency objectives to the teams. It is pointless to identify that user cases or code areas will be the most cost-critical without telling the developers this information.

Measure and test to ensure you are getting the performance and efficiency you want. Use the data from this to refine what changes need to be made to improve efficiency. As Dr Werner Voges says, as part of his Frugal Architect laws , cost optimisation is incremental.

“The pursuit of cost efficiency is an ongoing journey. Even after deployment, we must revisit systems to incrementally improve optimization. The key is continually questioning and diving deeper. Programming languages provide profiling tools to analyse code performance, and while these require setup and expertise, they enable granular analyses that can lead to changes that shave milliseconds. What may seem like small optimizations accumulate into large savings at scale.”

I discuss benchmarking and monitoring in **Chapter 9: Checking you are getting it right?**

Cautionary Tale

A company didn’t do any cost analysis and migrated all the data from their on-premises system to the cloud. The data was stored on expensive disc rather than object storage. The reality was very few customers used any data over two years old. Had they identified this before migration, they could have archived the data to cheaper storage or rewarded customers for deleting data.

Golden Rule

Don’t let procrastination stop you from making decisions but also don’t just make design decisions. Do a bit of analysis to support them!

https://www.thefrugalarchitect.com/laws/cost-optimization-is-incremental/

Chapter 5
What Shall I Code In?

"It is in your moments of decision that your destiny is shaped." **- Tony Robbins**

In this chapter, I want to address the subject of language choice. This is a tricky subject for many reasons. Firstly, many organisations have defined policies around the languages they will use. Stepping out of that box may not be allowable. Secondly, a more efficient programming language is not a panacea; inefficient code can be written in an efficient language. Thirdly, sometimes the code choice is irrelevant.

However, what is the most efficient programming language?

To be honest, I don't know. This is an area of research that needs more focus, but I will touch on it briefly below.

Exploring efficient language – an academic point of view

The most commonly referenced paper I see on social media is "Energy Efficiency Across Programming Languages: How Do Energy, Time, and Memory Relate?" by Pedro R. Pereira, Marco Couto, et al in 2017 . The paper investigates the relationship between energy consumption, execution time, and memory usage across 27 programming languages. The authors sought to provide empirical data to inform developers on how language choice impacts software efficiency. Using a set of well-defined benchmarks, they measured each language's performance in terms of time, memory, and energy usage.

https://www.thefrugalarchitect.com/laws/cost-optimization-is-incremental/

The study found significant differences between languages, with lower-level languages like C, C++, and Rust typically performing better in terms of energy efficiency, time, and memory usage, compared to higher-level languages like Python and Perl. However, if you are looking for a clear winner we are out of luck. Overall, C was the fastest and used the least energy, but Pascal used the least memory.

An important insight from the research is that faster execution times do not always correlate with lower energy consumption. Some languages, while fast, consume more energy due to factors like inefficient memory access patterns or garbage collection processes. For example, Java may perform relatively well in terms of execution time, but its memory management features, such as garbage collection, can result in higher energy usage. Conversely, some languages may take slightly longer to execute a task but consume less energy due to more efficient memory usage and resource management. **This challenges the common assumption that optimising for speed will inherently reduce energy consumption.**

The study emphasises the need for developers to balance energy, time, and memory efficiency based on the specific requirements of their project. While low-level languages like C and Rust offer the best overall balance of performance, developers working on higher-level languages may still need to optimise energy efficiency, particularly in contexts like mobile devices or data centres where power usage is a major concern.

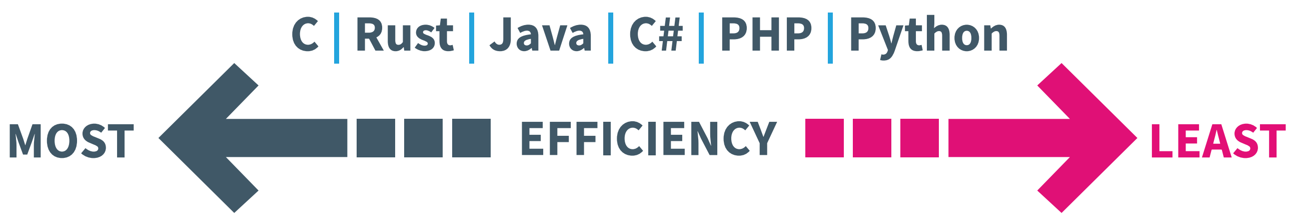
The **key takeaway is that the most energy-efficient language is not always the fastest,** and careful consideration of all three metrics — time, memory, and energy — is essential for efficient

software development.
You will have the same issue from a cost perspective: Fast does not always mean efficient. A faster language may use more resources than a slightly slower language, hence “cost” more.

Of course, low-level languages (C, Rust), aka compiled languages, are often the most cost-efficient. The hard work of translating the code down to the processor instructions (machine code) is done once at compile time. These languages often require the programmer to manage memory and system resources directly. Because of their proximity to the hardware, they offer better performance and control over how the machine operates, typically making them suitable for performance-critical applications. However, low-level languages can be more challenging to write and maintain due to their complexity and lack of abstraction.
High-level languages (Python), aka interpreted languages. This is where a program directly executes the instructions of a high-level programming language by translating them line-by-line into the processor instructions without needing to compile the entire program first. This process typically adds an overhead in CPU and memory costs, hence the cost. However, they provide greater abstraction from the machine’s hardware, allowing developers to write code in a way that is often more intuitive and closer to human languages. They manage many low-level details like memory allocation, system resources, and hardware interaction behind the scenes, which makes them easier to write, read, and

maintain.
There are languages like Java/C#, which are a sort of hybrid; they get compiled to byte code that is then interpreted. Techniques like Just In Time Compilers can be used. This is where the first time code is interpreted. However, if the code is frequently executed, then it is compiled to machine code. This is a great feature but does make it a challenge when benchmarking and testing **(see Chapter 9: Checking you are getting it right)**
Of course, what complicates this is that you can use code libraries in interpreted languages that are pre-compiled, and hence, some parts of your code can run as efficiently as compiled languages. For example, using the NumPy library in Python for matrix calculations is significantly faster than the native code.

Here is a cost-efficient chart, as yet not drawn to scale. Please do your own investigation.



Cautionary Tale

A company was worried about their spending on a particular serverless function. The cost model for these calls was based on the memory used and the duration of the function. The engineer felt that it was down to the function being written on C#, and they opted to re-write the function in Rust as it was more efficient. However, the savings were minimal. That was because, for that particular function, the majority of the time was spent calling and waiting for a response from a database server! When analysed, the C# code was responsible for 10% of the overall response time, so any improvement to it made little impact on the cost.

Golden Rule

Remember the overhead of programming in a more efficient language must save more than any increase in development and support costs. You also need a supply of developers that know the language!

Chapter 6
Code to Avoid Wastage

"The most cost-efficient code is often the code you don't write."

- Andrew Lee

Please excuse the self-promotion of this quote, but I want to make the point that every feature you code is likely to have a cost in the cloud. Some will be low, and some will be high. The key is making sure you are prepared to question the value!

Of course, there are other code-related changes you can implement that will help with cost control. Many of these techniques are core to building cloud native systems. Cloud native systems are software applications that are built to run in a cloud computing environment. Cloud native applications are designed to be scalable, resilient, and easy to manage.

Below are some of the techniques you can code into your software to avoid wastage and make it cheaper/easier to manage.

1. Load shedding

- Definition: Load shedding involves temporarily offloading non-essential processes during peak demand or high-cost periods to minimise overall expenses.
- Benefit:
  - By offloading or delaying less critical processes, organisations can reduce the total compute and storage resources consumed, thereby lowering their cloud costs.
  - This approach is particularly beneficial during sudden spikes in demand, which can lead to unanticipated cost increases if not managed effectively.

o Load shedding ensures that systems remain responsive for high-priority tasks while deferring less critical activities to less costly times.

2. Prioritisation of gold paying users

■ Definition: Not every user in your system will pay the same or have the same service level. Writing your code to prioritise/schedule these users avoids the situation where all users are treated equally in the system and additional resources are required to offer that level of service to all.

■ Benefit:
- o Guarantees that mission-critical processes (e.g., financial transactions, data writes) are not impacted during resource contention, maintaining high service quality and reliability.
- o Enables better control over resource allocation by distinguishing between essential and non-essential activities, reducing unnecessary consumption of expensive resources.
- o Creates an opportunity to reduce costs without compromising user experience, as essential functions are protected even during cost-cutting measures.

3. Leveraging spot pricing with stop and resume processing

■ Definition: Cloud providers offer spot instances at significantly reduced prices for spare capacity. However, these instances can be terminated with short notice, making them ideal for interruptible workloads.

■ Benefit:
- o The use of spot instances can reduce compute costs by up to 90% compared to on-demand instances, which makes it an attractive option for batch processing or non-urgent computational tasks.
- o By designing software that can gracefully handle interruptions — saving state and resuming processing as required — organisations can optimise for the lowest possible cost while maintaining operational integrity.
- o Spot pricing enables more granular control over spending, as processes can be dynamically paused and resumed based on real-time pricing fluctuations and resource availability.

4. Load throttling to avoid over-provisioning

■ Definition: Load throttling involves dynamically regulating the rate of incoming requests or transactions during periods of high demand to prevent the system from overloading resources.

■ Benefit:
- o Load throttling allows software to handle peak loads without requiring excessive provisioning of resources, which would remain underutilised during normal operation.
- o By controlling the rate of incoming tasks, the system can prevent performance degradation while staying within cost-efficient resource limits.
- o Reduces the need for upfront investment in high-capacity infrastructure and ensures that resources are right-sized to meet average rather than peak demand.
- o Load throttling also allows more predictable performance and cost management during demand fluctuations, providing a buffer against sudden surges without additional expenses.

5. Shifting non-critical workloads to off-peak periods

■ Definition: Non-critical workloads, such as data backups, batch processing, or analytics tasks, can be scheduled to run during off-peak hours when resource utilisation and demand are low.

■ Benefit:
- o Maximises the usage of available resources by shifting less urgent tasks to times when the system would otherwise be underutilised.
- o Reduces the likelihood of resource contention during peak hours, thereby enhancing the performance of critical workloads.
- o Off-peak scheduling allows organisations to take advantage of lower-cost resources or spot pricing during non-peak times, further optimising overall cloud spend.
- o By balancing the workload across different time periods, businesses can ensure a more stable and predictable

resource usage pattern, leading to better overall system efficiency and cost savings.

6. Data archiving and deletion policy

- Definition: A data archiving and deletion policy involves systematically identifying, archiving, or deleting data that is no longer actively used or required. This can include old log files, stale transactional data, or backups that have surpassed retention policies.

- Benefit:
    - Cost reduction: Reduces storage costs by minimising the amount of data retained in expensive storage solutions. Organisations can achieve significant savings by archiving less frequently accessed data to cheaper, lower-tier storage.
    - Improved performance: Reducing the active data footprint can enhance the performance of databases and applications by lowering the time it takes to process and retrieve information.
    - Regulatory compliance: A defined data archiving and deletion policy ensures that data retention aligns with legal and regulatory requirements, reducing the risk of non-compliance penalties.
    - Simplified maintenance: Regular data cleanup minimises clutter and simplifies the management and maintenance of cloud resources, enabling better monitoring and cost control.
    - Enhanced security: Deleting outdated or unused data reduces the potential attack surface, improving overall security posture by minimising exposure to vulnerabilities and reducing the impact of data breaches.

7. Dynamic Resource Allocation and Elasticity

- **Definition:** Cloud-native software is designed to automatically scale resources up or down based on real-time demand and usage patterns, ensuring optimal resource utilisation.

- **Benefit:**
    - Elastic scaling allows software to adapt to changing loads, minimising over-provisioning and the associated costs of unused capacity.
    - By dynamically allocating resources, organisations can maintain optimal performance during peak times without incurring unnecessary costs during low-demand periods.
    - This adaptability is crucial for managing variable workloads, and it also supports leveraging cost-effective resources, such as spot instances, whenever possible.

Cautionary Tale

While doing a cost optimisation exercise, an engineer discovered that the sizing of the database (and hence the costs) was driven by a few demanding SQL queries. The queries themselves were pretty optimal. It was just that as the company grew its customer base, they disproportionately became more demanding. This was because they were creating friend-type recommendations, and as more people joined, the complex connection logic would run longer.

The code ran for every login and featured the recommendations on the portal page hoping to encourage people to click through on the recommendations. Delving deeper into the web analytics, they noticed that hardly anyone used the functionality as it wasn't a core feature; in discussion with the business, it was decided to disable this feature to save money!

Golden Rule

The features you add and remove that make your code efficient are as important as the way you write them.

Chapter 7
Getting the Algorithm Right

There's a well-known joke about a tourist in Ireland who asks one of the locals for directions to Dublin. The Irishman replies: 'Well sir, if I were you, I wouldn't start from here'.

Algorithm design plays a fundamental role in determining the efficiency and performance of computer programs. A well-designed algorithm can drastically reduce the time and resources required to solve a problem, while a poorly designed one can make even simple tasks computationally expensive. The impact of good algorithm design is so significant that it often determines the feasibility of solving complex problems within a reasonable timeframe. This is where concepts like asymptotic notation, particularly Big-O notation, become critical for understanding and comparing the efficiency of different algorithms.

Understanding algorithm efficiency

When discussing algorithm efficiency, we are primarily concerned with how the amount of work (e.g., time or memory) required scales with the size of the input. This scaling behaviour is captured using Big-O notation, which describes the upper bound of an algorithm's growth rate.

For example:

- An algorithm with a time complexity of O(n) means that the time it takes to execute grows linearly with the size of the input.
- An algorithm with O(n2) complexity means that the time required increases quadratically with input size, i.e. doubling the input size would increase the time by a factor of four.

Why Big-O Notation Matters

Big-O notation provides a way to express how much more work an algorithm requires as the input size increases. For instance, consider two algorithms for summing the elements in a list of numbers:

1. Algorithm A has a time complexity of O(n).
2. Algorithm B has a constant time complexity of O(1).

For small inputs, both algorithms might perform similarly. However, as the input size increases, Algorithm B, with its O(1) complexity, will become significantly more efficient compared to Algorithm A.

The following table illustrates this difference:

Input Size (n)	Algorithm A: O(n)	Algorithm B: O(1)
10	10 operations	1 operation
100	100 operations	1 operation
1,000	1,000 operations	1 operation
10,000	10,000 operations	1 operation

The values in the table show the number of operations each algorithm would perform for different input sizes. As the input size increases, Algorithm A's work grows linearly, while Algorithm B's work remains constant, demonstrating a significant reduction in computational effort due to better algorithm design.

Example: Summing the Elements in a List

Let's consider a concrete example to illustrate how different algorithm designs can impact the efficiency of solving a problem.

Problem: Given a list of numbers, find the sum of all elements.

Naive Algorithm: Use a loop to iterate through each element and add them together.

```
public class SumNaive {
 public static int sumNaive(int[] arr) {
  int total = 0;
  for (int num : arr) {
   total += num;
  }
  return total;
 }

 public static void main(String[] args) {
  int[] numbers = {1, 2, 3, 4, 5};
  System.out.println(“Sum: “ + sumNaive(numbers)); // Output: 15
 }
}
```

Time complexity: This approach iterates through each element once, resulting in a time complexity of O(n).

Optimised algorithm: Use the mathematical formula for summation of an arithmetic series:

For a list containing numbers from 1 to n, the sum can be calculated using the formula:

$$Sum=\frac{(n(n+1))}{2}$$

```
public class SumOptimized {
 public static int sumOptimized(int n) {
  return n * (n + 1) / 2;
 }

 public static void main(String[] args) {
  int n = 5; // Sum of numbers 1 to 5
  System.out.println("Sum: " + sumOptimized(n)); // Output: 15
 }
}
```

- **Time Complexity**: This formula uses only a few arithmetic operations, regardless of the input size, resulting in a constant time complexity of O(1).

For a list of 1,000,000 elements, the naive algorithm would require 1,000,000 operations, while the optimised algorithm would need only a single operation — a dramatic reduction in computational work.

The importance of algorithm design in practice

Algorithm design and analysis are not just academic exercises. They have practical implications for a wide range of applications, from data processing to machine learning, database management, and more. For instance, search engines rely on sophisticated algorithms to quickly retrieve relevant information from vast amounts of data, while encryption algorithms ensure secure communication by making decryption infeasible within reasonable timeframes.

Conclusion

Algorithm design is a powerful tool for optimising code and reducing computational work. By leveraging techniques such as asymptotic analysis and Big-O notation, developers can choose or design algorithms that scale well with input size, thus minimising computational resources. In practice, selecting an appropriate algorithm can transform an otherwise infeasible problem into one that can be solved efficiently, making algorithm design an essential skill for programmers and computer scientists alike.

Cautionary Tale

While choosing algorithms with better asymptotic complexity is generally beneficial, it’s important to consider that algorithms with lower theoretical time complexity are not always faster for small inputs in real-world applications. This is due to the constant factors and overhead involved in some optimised algorithms.

Golden Rule

Always aim for the simplest algorithm that achieves the desired outcome with the least amount of work.

This means choosing an algorithm that solves the problem correctly while minimising computational resources, ensuring efficiency.

Chapter 8
Crafting Cost Efficient Code

"More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason — including blind stupidity."

— William A. Wulf

When it comes down to crafting lines of code for efficiency, the mantra is less is more. That is, less resource used (CPU/Disk/Storage etc.) means more cost savings. You must think about the concept of work i.e. the amount of CPU cycles or data that needs to be processed. The more you can write your code to do the required functionality in the least amount of CPU cycles or memory, the better.

Below are eight techniques for minimising the work you need to do. Remember, sometimes there is never a single solution

1. Reduce the amount of work or data you store. This is probably the key optimisation that you can do. This could be a fairly simple optimisation. For example, using Precomputation to calculate frequently used values ahead of time and storing them to avoid repeated computation during runtime. This is particularly powerful when used to optimise loops (known as hoisting). This can be seen in the loop example below:

```
int factor = x * y; // Factor is calculated once before the loop
  for (int i = 0; i < numbers.length; i++) {
   sum += numbers[i] * factor;
  }
```

Another example may be as simple as using short-circuiting. This is for logical operators, where the execution will skip the processing of the subsequence operands if any fail. This is achieved in some languages using (e.g., `&&` or `||` in C-style languages). Hence, if you look at the code snippet below, then in that case, it would output 4 as the first conditional is false, and the second is not executed.

```
public static void main(String[] args) {
 int a = 10;
 int b = 4;

 if (a == 0 && ++b == 5) {
  //Do Stuff;
 }
 System.out.println(b);
```

2. Be careful of the trade-off between data and compute. You may save memory space by encoding data but spend more CPU computational time encoding and decoding it. You have to decide what will be the biggest cost dimension for this. For example, if you have lots of data but do very little processing, then encoding that data will be better.

3.Reuse things that you would have to compute again. This is more of a tricky optimisation as memory costs, so you have to get benefit from anything you save. Also, caching data may benefit response time! Again, you are in the cost vs. response time trade-off. However, common-subexpression elimination helps with coding efficiency.

4.Exploit. If libraries or native functions exist to do things you need to do, check them out. They may very well be faster than the code you can write. This is particularly true with higher-order languages that are interpreted, as libraries may be directly pre-compiled and run super efficiently. Java provides a variety of optimised methods in the Java.util package, such as Collections.sort() for sorting lists.

5.Sympathise. There is the concept of Mechanical Empathy. This is originally a quote from British Racing Driver Jackie Stewart: "You don't have to be an engineer to be a racing driver, but you do have to have Mechanical Sympathy". This is about writing code that makes the most of the underlying hardware.
For example, can writing your code to maximise the process cache improve efficiency? To illustrate the effect of the cache, consider the task of summing all the elements in a 2D matrix. We simply loop through each element, adding the element's value to the sum. We have a choice to iterate row by row or column by column. The code to iterate row by row is shown below:

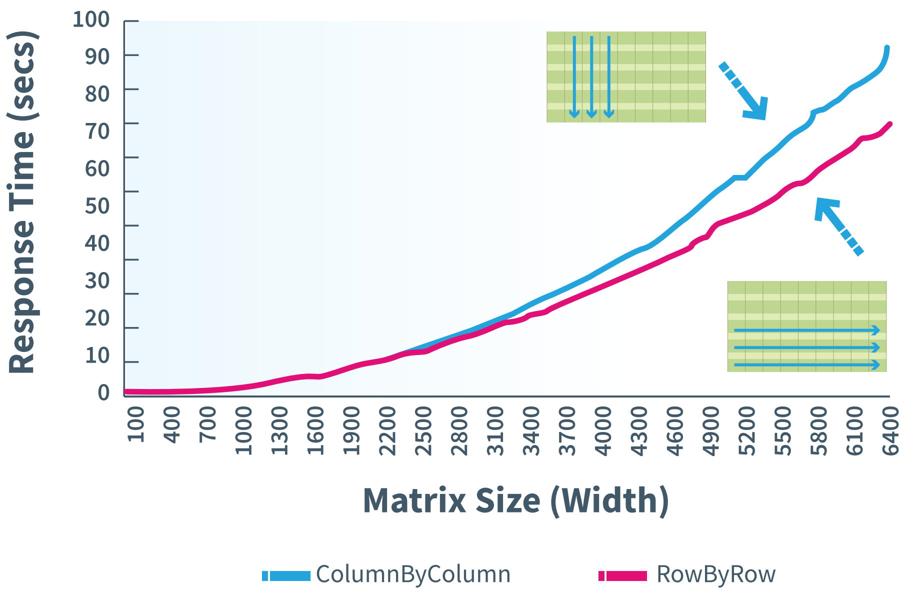
```
for i in range(Size):
    for j in range(Size):
        sum += x[i][j]
```

To change it to column by column, you would just adjust the array indexing to:

```
sum += x[j][i]
```

Now, we would expect that regardless of the iteration choice, the programs should take an equal amount of time. However, when I ran tests for various matrix sizes, there was a distinct difference in the execution time.

Processor Cache in Action



So, why do we observe different performance? It has to do with how memory is loaded into the cache. After the first row is loaded into the cache, it prefetches the next row. Since row-by-row iterations match this prefetching, more data can be retrieved from the cache, thus improving performance. Additionally, the cache is 64-bit, so if each element of the matrix is smaller than this, you will get multiple elements per cache line.

It's interesting to note, one thing that is often forgotten is that retrieving memory from the CPU cache or main memory is counted as part of CPU time.

6. Match the data structure you choose to the operations you are doing. For example, you could store data in an array, list or a Hashmap. Each of these offers different efficient characteristics depending on what type of operations and what dimension you want to be efficient. For example, the most efficient for an index lookup could be an array when compared to the others but an array needs to be fixed size. This means that if the array is sparsely populated you will not be memory efficient. Again, this is a case of understanding the usage(workload) and requirements to choose the best fit.

7. Predict. Write your code in the order you think it will get executed to aid branch prediction. Writing code in a way that aids the CPU's branch prediction mechanisms can lead to significant performance improvements. This often involves structuring if-else blocks such that the most likely branches are evaluated first. Even if you are not worried about branch prediction on the CPU the idea of ordering tests is to perform those that are more often successful. Similarly, inexpensive tests should precede expensive ones.

Inefficient code by not combing the tests

```
if (age >= 18) {
 if (salary >= 40000) {
   if (hasGoodCredit) {
   System.out.println("Eligible for a loan.");
    }
  }
 } else {
   System.out.println("Not eligible for a loan.");
 }
```

Combing tests and using short circuiting for improved efficiency

```
 if (age >= 18 && salary >= 40000 && hasGoodCredit) {
   System.out.println("Eligible for a loan.");
  } else {
   System.out.println("Not eligible for a loan.");
  }
```

8. Test / experiment. Now this is not easy. If you are really looking at optimising code, test your trade-offs (see the next chapter). Also, don't underestimate the power of explaining your code to another developer. You will probably spot optimisations if you talk it through!

Cautionary Tale

A development team combined Java functions with the aim of reducing the overhead of calling the function. However, in the test, this had a negative effect as Java's Just-In-Time (JIT) compiler can optimise small functions by inlining them. In addition, this also improves the cache hit ratio on the processor. (Good job they tested their code!).

Golden Rule

Remember, to optimise what needs to be optimised not what you want to optimise

Chapter 9
Checking You Are Getting It Right

"Programming today is the opposite of diamond mining. In diamond mining you dig up a lot of dirt to find a small bit of value. With programming you start with the value, the real intention, and bury it in a bunch of dirt."

— Charles Simonyi

During the development cycle, you will need to determine if certain pieces of code are efficient enough, or you may need to benchmark code to help you make the right decisions. These small developer-led benchmarks are called **micro benchmarks.**

Typically, micro benchmarking involves measuring the performance of very small sections of code, usually individual functions or small code blocks. Micro benchmarking helps identify inefficiencies in specific parts of code, enabling developers to optimise critical sections. It provides insights into how various algorithms and data structures perform, especially under different scenarios or inputs. This allows for comparison of different implementations, be it a function or algorithm or maybe which library to use. A mature organisation may incorporate this into their development pipelines to help detect performance and efficiency.

If you are going to benchmark, the trick is getting it right, such that the decisions made when running code locally in a development environment work just as well in a production environment with real world data.

Common pitfalls of micro benchmarking

1. Lack of context: Micro benchmarks measure the performance of isolated code segments, often ignoring broader context, such as memory hierarchy effects, network latency, or interactions with other parts of the application.

2. Compiler optimisations: Modern compilers can optimise code in surprising ways. For example, the compiler might remove or change certain code paths when it determines that the results are not used. This can lead to inaccurate measurements. For example, Java uses Just in Time compilation, and this may or may not occur when you run your benchmark in the same way as production.

3. Unstable measurements: Small code segments can produce noisy results due to variations in system state (CPU frequency scaling, background tasks, etc.). Ensuring stability and consistency in measurements can be challenging.

4. Hardware and OS dependencies: The results of a micro benchmark can vary widely based on the underlying hardware (CPU, memory, cache) and the operating system. Benchmarks on one machine might not generalise to others.

5. Ignoring Real-World Scenarios: Micro benchmarks often use idealised inputs and configurations that do not represent typical usage patterns, leading to misleading conclusions about actual performance.

As you can see above there are a lot of bear traps to grab you when you execute micro benchmarks. So, what can you do to help avoid these?

- Run multiple iterations: Run the benchmark many times and calculate statistical measures like mean, median, and standard deviation to get reliable results.

- Use appropriate tools: Use dedicated benchmarking libraries or tools like Google Benchmark for C++, Benchmark.js for JavaScript, Java Microbenchmark Harness for Java or 'timeit' for Python to minimise measurement overhead.

- Analyse in context: Use micro benchmarking in conjunction with profiling and macro-level performance testing to get a holistic view of performance.

The last one is one of the most tricky as performance and efficiency are not always the same. You need to look into the underlying execution to determine the efficiency.

To give some context there is the Processor Iron Law.

The **Processor Iron Law** is a principle in computer architecture that describes the trade-off between execution time, the number of instructions executed, and the clock cycle time (or how fast instructions are processed). It states that the execution time of a program is determined by three factors: the number of instructions executed, the average number of clock cycles required per instruction (CPI), and the clock cycle time. Mathematically, it's expressed as:

Execution Time = Instructions × CPI × Clock Cycle Time

This law highlights that improving overall execution time typically involves optimising one or more of these variables. For example, reducing the number of instructions by using more efficient algorithms, decreasing CPI through better instruction scheduling or pipelining, or decreasing the clock cycle time by using faster processors can all lead to faster program execution. However, changes in one area can affect the others, making it crucial to balance these factors when designing or optimising a system.

Hardware performance monitoring unit (PMU) counters

Luckly, there are hardware performance monitoring unit (PMU) counters which are specialised registers provided by modern CPUs that track various low-level hardware events, such as cache misses, branch mispredictions, and instruction counts. They are invaluable for understanding the underlying performance characteristics of code, especially in the context of micro benchmarking. A good micro benchmarking tool will collect and present PMU counters.

Key PMU Metrics will measure:

- Instruction count: Track the number of instructions executed to understand the efficiency of the code. The fewer instructions, the better.
- Cache usage: Monitor cache hits and misses to identify memory access patterns and locality issues. Maximising cache hits will improve efficiency, but beware: this behaviour will dramatically change in a production system with multiple code paths running.
- Branch performance: Use branch prediction counters to see if the code is causing frequent mispredictions. The less misprediction, the better.
- Stalls and pipeline issues: Measure stalls (e.g., memory stalls, instruction fetch stalls) to see if the CPU pipeline is effectively utilised.

Running different micro benchmarks allows you to see the impact of the hardware and helps you make decisions about optimal code choice.

Cautionary Tale

A software developer micro benchmarked several versions of their new function. Ignoring the PMU counters they chose the fastest. However, this was less efficient than other options and led to additional costs in the cloud.

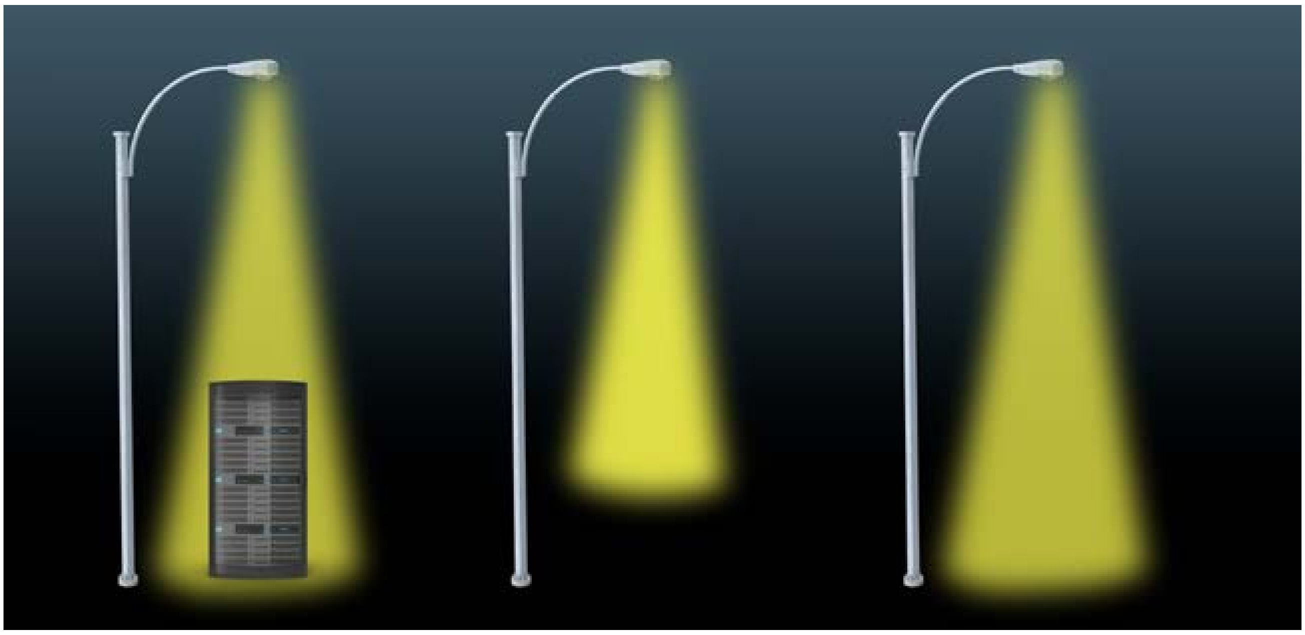
Golden Rule

Be driven by measurement, but make sure what you measure is representative of real-world scenarios.

Chapter 10
Checking You Got It Right? You Won't

"The streetlight effect occurs when people only search for something where it is easiest to look."

- David H. Freedman



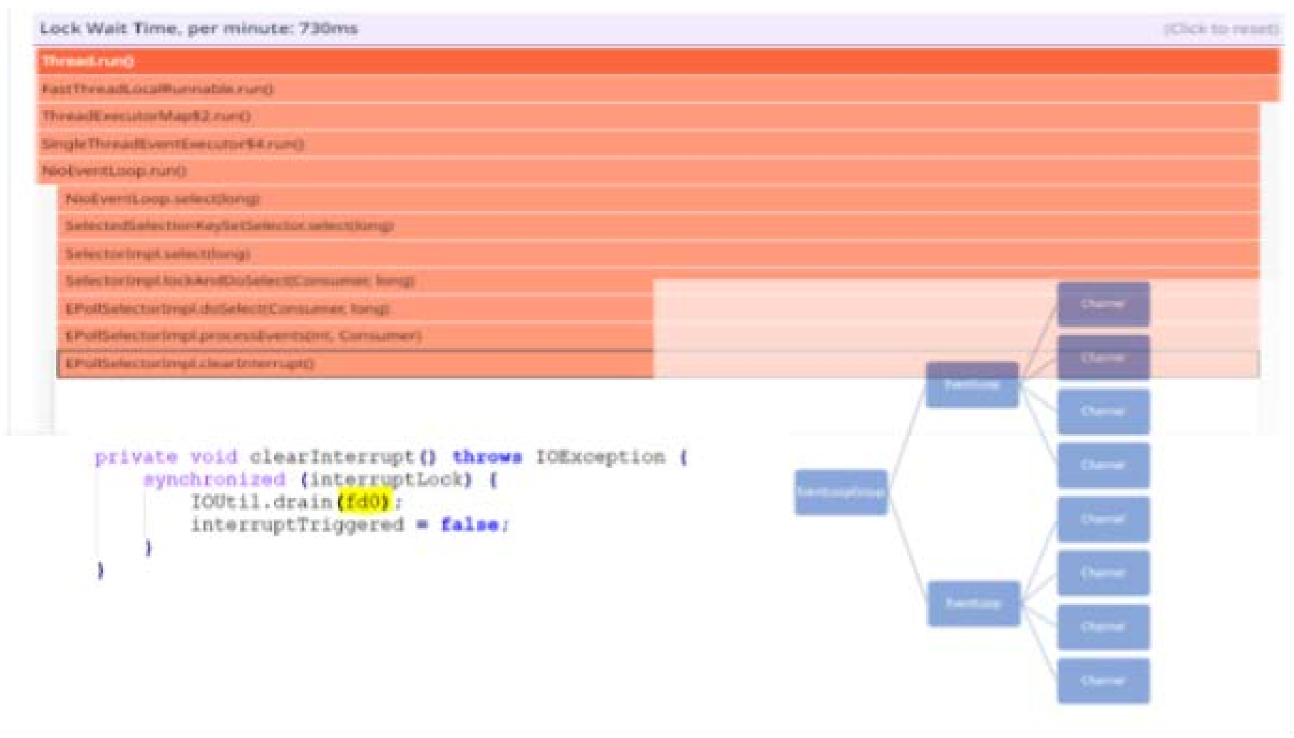
When trying to identify areas to optimise in cloud-based systems, the primary challenge is achieving visibility into the various components to determine if they are running efficiently. This is where observability and cloud monitoring tools play a critical role. These tools provide insights into resource utilisation, performance bottlenecks, and application behaviour, allowing you to spot inefficiencies.

One approach to resource optimisation is to ensure that your cloud resources are operating within the "Goldilocks Zone" — not over-utilised to the point where response times are impacted and not underutilised, leading to unnecessary costs. If you remember the graphic in **Chapter 3**, it shows how response time drastically degrades after 70%-80%. For instance, you can monitor CPU, memory, and I/O to ensure that the resources are not being throttled or sitting idle.

However, merely having a resource at an ideal utilisation level is only the initial step in achieving efficiency. The next phase involves verifying that the software executing on those resources is configured and behaving in an optimal manner. Just think; a poorly written, inefficient code will use a lot of computing resources. This can be a complex process because even if the hardware utilisation appears to be balanced, inefficiencies in code execution, suboptimal algorithms, or improper configurations can lead to hidden performance and cost inefficiencies.

Using flame graphs to detect CPU usage and pinpoint optimisation areas

Flame graphs are a powerful visualisation tool that can help detect inefficiencies at the CPU level. They provide a detailed view of where time is being spent in your code, making it easier to identify performance bottlenecks, hotspots, or sections of the code that may need optimisation. By visualising CPU activity, flame graphs can show you if certain functions or threads are over-utilising CPU resources, potentially leading to performance degradation.


Lock Wait Time, per minute: 730ms
(Click to reset)
Thread.run()
FastThreadLocalRunnable.run()
ThreadExecutorMap$2.run()
SingleThreadEventExecutor$4.run()
NioEventLoop.run()
NioEventLoop.select(long)
SelectedSelectionKeySetSelector.select(long)
SelectorImpl.select(long)
SelectorImpl.lockAndDoSelect(Consumer, long)
EPollSelectorImpl.doSelect(Consumer, long)
EPollSelectorImpl.processEvents(int, Consumer)
EPollSelectorImpl.clearInterrupt()
EventLoopGroup
EventLoop
EventLoop
Channel
Channel
Channel
Channel
Channel
Channel
Channel
Channel

```
private void clearInterrupt() throws IOException {
    synchronized (interruptLock) {
        IOUtil.drain(fd0);
        interruptTriggered = false;
    }
}
```

For example, flame graphs allow you to see which functions are consuming the most CPU time, indicating areas where code improvements can yield significant gains. This makes it easier to focus optimisation efforts where they will have the most impact, such as optimising or re-writing specific algorithms, minimising I/O wait times, or parallelising operations to distribute the load more effectively.

Importance of cost allocation and tagging

In addition to optimising performance, managing cloud costs is equally crucial for efficient cloud operations. Cost allocation and tagging are fundamental practices that enable you to understand and manage cloud expenses effectively. Tagging resources is akin to labelling expenses in a budget. By tagging resources with metadata such as project names, department codes, or environment identifiers (e.g., "dev" or "prod"), you gain the ability to track costs, identify spending patterns, and ensure accountability within your organisation.

Ideally, proper tagging should enable the calculation of ****cost per transaction****. This requires a clear understanding of both resource usage and demand. For instance, by tagging resources based on different transaction types or services, you can attribute costs to the number of transactions processed. This approach allows you to monitor costs relative to business activity and ensure that expenses are proportional to transaction volumes. Without consistent and meaningful tagging, it becomes challenging to measure efficiency, as you cannot correlate usage with transaction counts.
By achieving cost-per-transaction visibility, organisations can make informed decisions on resource allocation, identify opportunities to optimise specific transactions, and forecast costs more accurately. Ultimately, this level of granularity provides transparency into cloud expenses and allows for better resource and budget management.

In summary, achieving cloud optimisation requires a holistic approach that combines resource observability, performance analysis using tools like flame graphs, and robust cost management through tagging. These practices ensure that your cloud infrastructure is not only performing optimally but is also cost-effective.

Cautionary Tale

A company did not have full coverage of monitoring their application in production. Without this, they often increased capacity (wasted costs) to avoid performance issues rather than solving the root cause of the performance issue.

Golden Rule

You can only effectively manage what you can monitor.

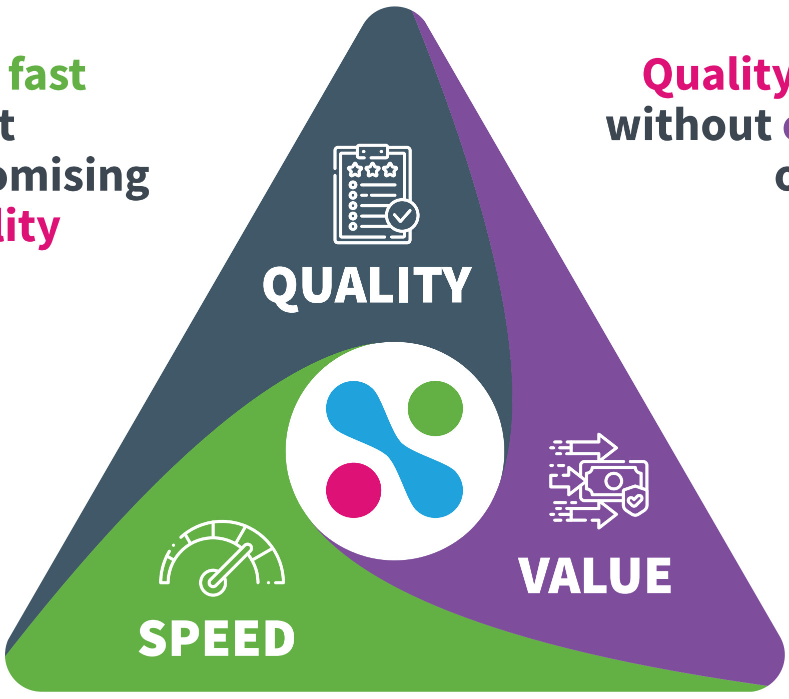
Chapter 11
What About the Soft Costs?

"Time is the scarcest resource, and unless it is managed, nothing else can be managed."

– Peter Drucker

Deliver fast without compromising on quality

Quality products without expensive overheads



Value for money delivery without slowing down

If you're working in the cloud, transitioning to the DevOps methodology is essential. The primary goal of DevOps is to improve product delivery quality by facilitating fast, end-to-end development cycles. This allows teams to deliver high-quality products rapidly while offering value for money. However, it's easy to lose sight of costs during this shift and, without proper oversight, you can accumulate significant expenses when implementing DevOps.

DevOps teams must prioritise cost efficiency. The mindset of the team should inherently include cost management to ensure sustainable growth and operational agility. Designing and deploying solutions with cost efficiency at their core will enable teams to optimise resource usage, reduce unnecessary operational expenses, and ultimately improve the return on investment. Cost efficiency is key to developing scalable, robust, and financially sustainable DevOps practices.

Automating environment builds

DevOps encourages the maximisation of automation tools, particularly for infrastructure provisioning. Managing cloud applications is a significant cost and automation is critical to easing this burden. Tools such as Infrastructure as Code (IaC) enable the definition and provisioning of cloud infrastructure using code.

IaC allows teams to replicate environments, track changes, and roll back configurations seamlessly, streamlining operations. For instance, teams can use IaC to create test environments that mirror production, spinning them up only when necessary. This reduces both operational complexity and cost.

Reducing human support costs

Your application architecture can benefit greatly by offloading operational responsibilities to managed platforms and services. These non-differentiated workloads are tasks that do not give your business a competitive advantage but are necessary for operations. Examples include databases, message queues, or content delivery networks (CDNs). These services are critical but do not differentiate your product from others in the market. By using managed platforms for these workloads, you free up your team to focus on core business innovations.

Examples of cloud services that manage non-differentiated workloads include:

- Amazon RDS (Relational Database Service) or Google Cloud SQL for database management
- Amazon SQS (Simple Queue Service) or Azure Queue Storage for message queuing
- Amazon CloudFront or Azure CDN for content delivery

By utilising such services, you reduce human intervention, streamline operations, and minimise the cost associated with managing infrastructure manually.

Cloud provider agnosticism

Adopting a hybrid or multi-cloud strategy can be a cost-effective approach, particularly if you remain cloud provider agnostic. This means that your systems are designed to run on any cloud provider, allowing you to choose vendors based on specific service costs or features. Often, smaller vendors specialise in niche services — such as storage or content delivery — and may offer cheaper alternatives compared to major cloud providers like AWS, Azure, or Google Cloud.

However, there are scenarios where sticking with a single cloud provider is advantageous. For example, if your application heavily relies on proprietary services like **Amazon Lambda** (for serverless computing) or **Azure Cosmos DB** (for globally distributed databases), using a single provider can streamline integration, simplify billing, and improve performance. Additionally, single-cloud solutions can enhance security through tighter controls and native support for regulatory compliance in certain industries.

In essence, cloud provider agnosticism offers flexibility and cost advantages, but in cases where deep integration with specific services or performance optimisation is crucial, relying on a single provider may provide a more seamless experience.

Implementing self-healing systems

Building self-healing and self-remediating cloud applications ensures resilience and minimises downtime. Automating the detection and resolution of common issues allows systems to automatically recover from failures or performance degradation. Using cloud-native tools, you can design applications that respond autonomously to incidents, maintaining operational continuity.

Instrumentation from the start

Instrumentation is essential for observability and continuous integration/continuous delivery (CI/CD) pipelines. Integrating instrumentation directly into the application from the beginning can reduce the need for expensive external monitoring solutions or sidecar containers. However, it's important to consider the storage and overhead requirements associated with the logging generated by your application.

Remember to balance maintainability vs optimised code

One common argument against heavy optimisation is the increased maintenance burden it may impose. However, not every line of code needs to be optimised for maximum efficiency. Focusing optimisation efforts only on cost-critical components is crucial to striking a balance between optimisation and maintainability.

Maintainable code should follow clean code principles, ensuring readability, modularity, and scalability. Clean code is easier to manage, debug, and extend, which reduces long-term maintenance costs. Moreover, managing **cognitive complexity** — the ease with which someone can understand the code — is vital. Excessive optimisation often results in convoluted code that's difficult to understand and maintain. Code with low cognitive complexity is simpler to manage and less prone to errors.

In summary, it's important to be agile as an organisation. Know when to prioritise efficiency and when to prioritise maintainability. Optimising only where necessary while adhering to clean code principles will ensure that your systems remain cost-effective, scalable, and easy to maintain over time.

Cautionary Tale

Optimised code is great unless the support team is debugging it at two in the morning as part of a critical incident. Be clever but remember to comment!

Golden Rule

Take a holistic approach to costs!

Chapter 12

Making It Stick

"Change is not just about adopting new strategies but creating a culture that sustains your purpose and vision for the long-term."

— John P. Kotter

This chapter takes it lead from Dr Manzoor Mohammed, **Cloud Cost Optimisation A more thoughtful approach download**

A well-defined strategy focused on cost optimisation is essential for becoming a cost-efficient organisation.

The approach centres around sustainable cost optimisation by analysing historical data, embedding efficient practices, and predicting future spending trends. Here at Capacitas, we base our methodology on the "Frugal Architect's" approach to software development, introduced by Werner Vogels, CTO at Amazon.

Our aim is to help organisations solve the problem of spiralling cloud costs, accelerate development, and harness the full potential of the cloud.

1. Awareness

It's crucial that teams understand how much they're spending on cloud services in relation to the overall IT budget. Focusing on spend as a portion of the IT budget provides a clearer picture, avoiding the false sense of efficiency that might arise from comparing it to overall business revenue. Teams should have visibility into past spending patterns and the ability to forecast future costs.

Beyond a high-level overview, engineers should understand how their services contribute to the total cloud budget and how their decisions impact service performance and reliability. This awareness helps teams more effectively leverage cloud cost management tools like Cloudability, CloudHealth, AWS Cost Explorer, and Azure Cost Management.

2. Prioritisation and time management

Teams must prioritise cloud cost optimisation activities, such as reducing unnecessary capacity or eliminating non-impactful costs. These activities should be integrated into sprint cycles or have dedicated cycles to ensure continuous focus. Housekeeping activities, often overlooked, are vital for understanding cloud costs and maintaining efficient operations.

3. Observability

Visibility into cost, performance, and utilisation data over both the short and long term is essential. This visibility reflects a team's confidence and understanding of their systems. For example, less confident teams might log everything, keep data longer than necessary, or rely heavily on logs instead of metrics, all of which can increase costs.

Teams should consider three key areas:

- Tagging: Properly tag cloud systems to track cost ownership.
- Metrics vs. Logs: Use metrics for easier analysis and reduced costs.
- Granularity and Data Retention: Maintain appropriate data granularity — one-minute for system data and one-hour for cost data.

4. Understanding

Teams need to understand how cloud costs relate to their services. They should be able to quantify and articulate the business drivers of capacity utilisation, cost, and performance. This understanding is particularly vital for data-driven workloads, where costs might increase more rapidly than revenue growth.

5. Confidence

Teams must have confidence in their ability to adjust cloud infrastructure without compromising service reliability. Excess capacity often stems from a lack of confidence in system reliability. By increasing confidence, teams can eliminate unnecessary capacity and deliver quality code faster.

6. Product Value

A strong grasp of the product's value to users and the business is key to controlling costs. Align business and engineering teams on the required performance levels and determine the relevance and obsolescence of product features to manage costs effectively.

7. Predictability

Teams should have a long-term view of cloud spend, predicting costs over the next three years based on business demand. This predictability aids in negotiating commitments with cloud providers and enables better conversations about the value delivered by cloud services.

These seven principles form the foundation of our thoughtful approach to cloud cost optimisation, helping our clients achieve long-term value from their cloud investments while minimising costs and enhancing performance.

Cautionary Tale

A company struggling with cloud costs decided to incentivise its development and platform team to save costs. They ran a competition between the teams, and the team that saved the highest percentage won a vacation to Hawaii. The initial savings were fantastic, but after the competition ended, they grew rapidly as the teams waited for the next competition!

Golden Rule

There is no silver bullet. For long-term, sustained cost savings and control, embed cost-efficiency into the company's psyche.

Conclusion

Remember, the cloud is still just offering compute power, but now you can rent that by the second! Or rent nothing when you don't need it.

The cloud is a great opportunity to drive down your organisation's costs. However, it is like having children: great fun but not always easy to get right and sometimes they will drive you to tears. (Some may be happy tears, others not!).

You have a lot to consider in any software project with many competing demands with added cost and time pressure. However, if you are going to realise the benefits and the ultimate cost saving in the cloud you need to design for cost at the very beginning.

Hopefully, this white paper will help you achieve this. You also have to remember everything is proportional this means not every line of code needs to be optimised. Not every subcomponent needs to be trimmed to the bone. The trick is setting off on the right path, making changes along the way and focusing on the components that will cost the most to run. Remain agile at all times.

Good luck

Andrew Lee

Andrew Lee is a highly experienced Performance Engineer with over 30 years of expertise in load testing, system modelling, cost optimisation and capacity planning on large-scale IT projects.

For the past 2.5 years at Capacitas, Andrew has helped customers meet their performance goals and reduce their cloud costs. Previously he was a distinguished engineer for a large international service provider working for customers across the global.

Andrew's deep technical expertise spans a range of performance engineering disciplines, from strategy and management to diagnostics and testing. He is adept at using a variety of industry-leading tools to identify bottlenecks, optimize system performance, and provide actionable insights.

With a proven track record in performance engineering, Andrew is passionate about driving efficiency, ensuring system resilience, and helping organizations achieve high-performing, cost effective scalable IT solutions. He often posts on linkedIn www.linkedin.com/in/andrewjohnlee/

CLOUD DONE CORRECTLY

capacitas®

COST | SCALABILITY | PERFORMANCE

www.capacitas.co.uk