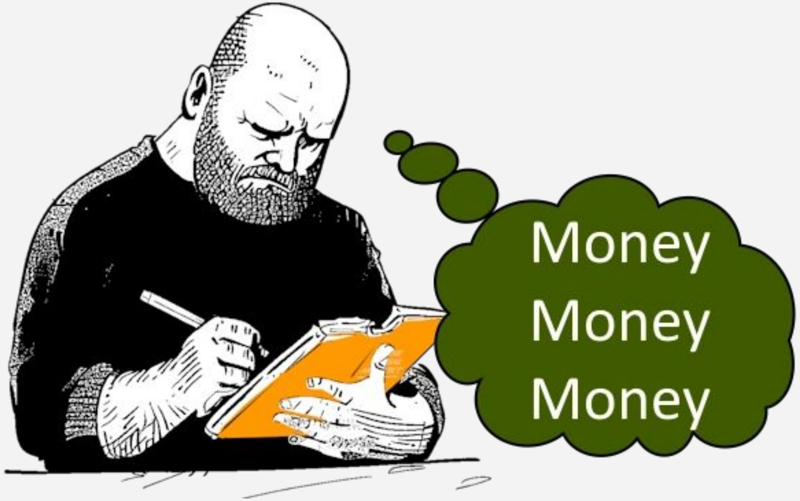
The Frugal Architecture in Practice



Artem Polishchuk

18 December 2024

About me

- Solution Architect at Ciklum
- 12 years of experience.
- Work more than 7 years with Cloud
- Frugal by default

My blog:

https://bool.dev/blog

My LinkedIn:

https://www.linkedin.com/in/apolischuk/

Agenda

01 What is the Frugal Architecture

02 Phase 1: Design

03 Phase 2: Measure

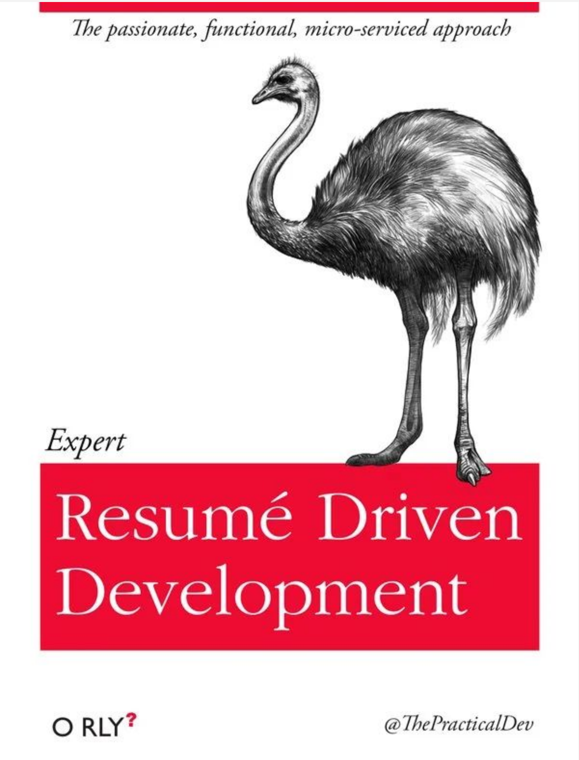
04 Phase 3: Observe

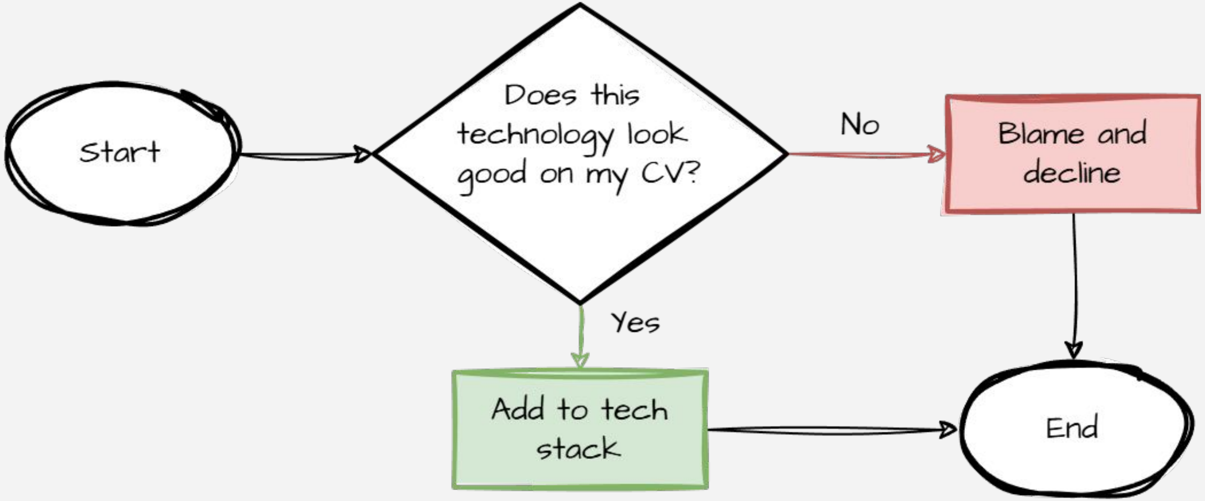
05 Common Pitfalls

06 Key takeaways

07 Q&A

Did you care about cost efficiency when you design your architectures?


The passionate, functional, micro-serviced approach
Expert
Resumé Driven Development
O RLY?
@ThePracticalDev



Start

Does this technology look good on my CV?

No

Blame and decline

Yes

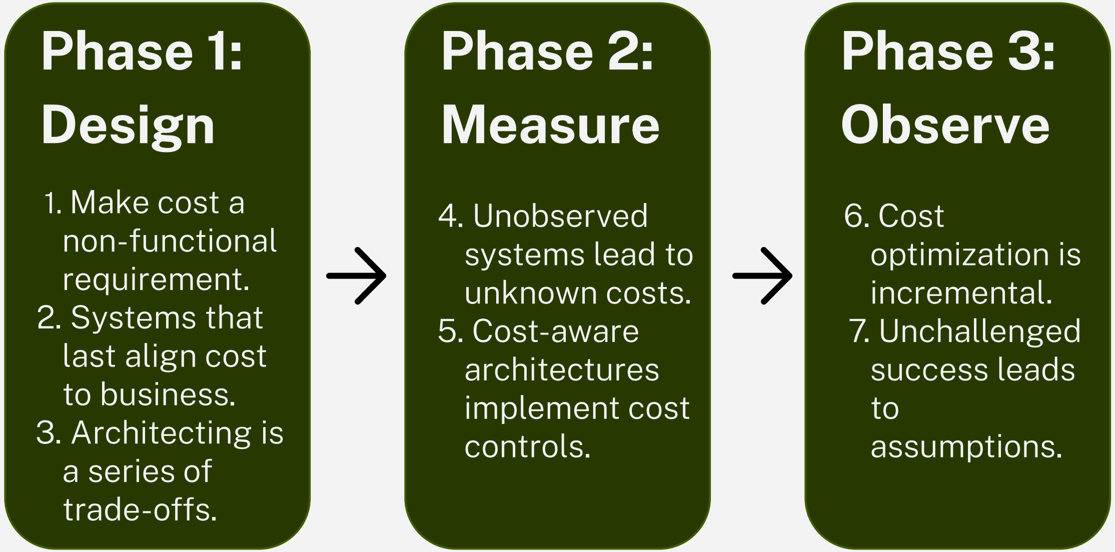
Add to tech stack

End

What is the Frugal Architecture?

(by Dr. Werner Vogels, CTO of Amazon)

> The Frugal Architecture approach advocates for a sustainable, cost-effective, and resource-efficient architectural design and implementation methodology.



Phase 1: Design

1. Make cost a non-functional requirement.
2. Systems that last align cost to business.
3. Architecting is a series of trade-offs.

→

Phase 2: Measure

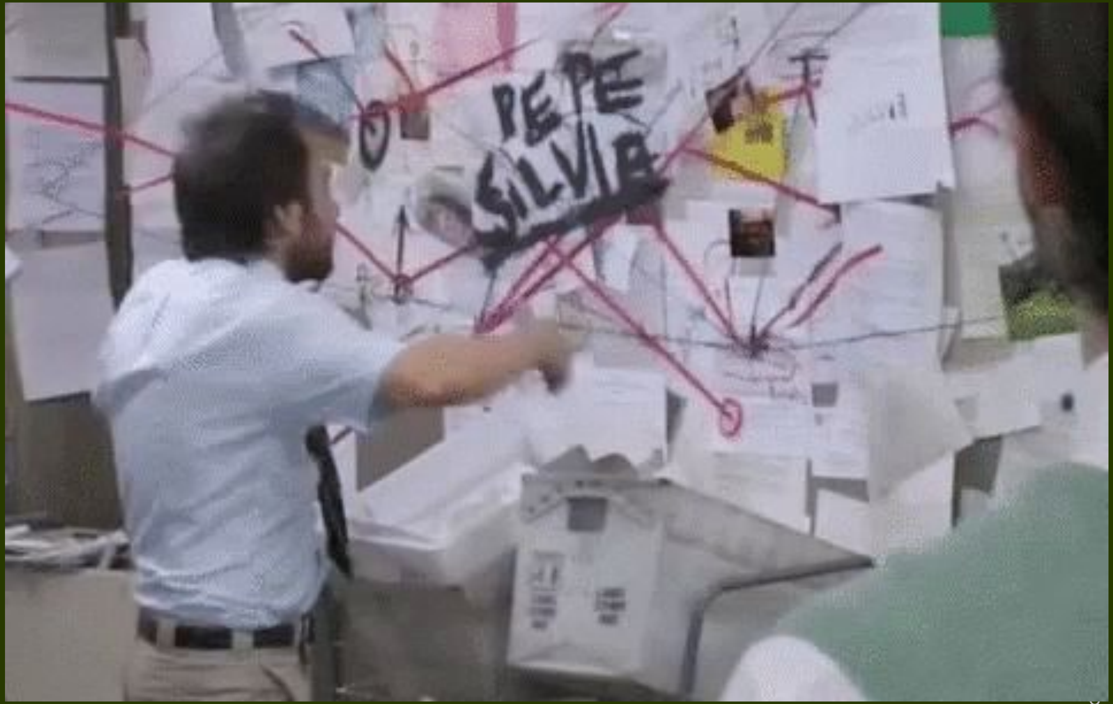
4. Unobserved systems lead to unknown costs.
5. Cost-aware architectures implement cost controls.

→

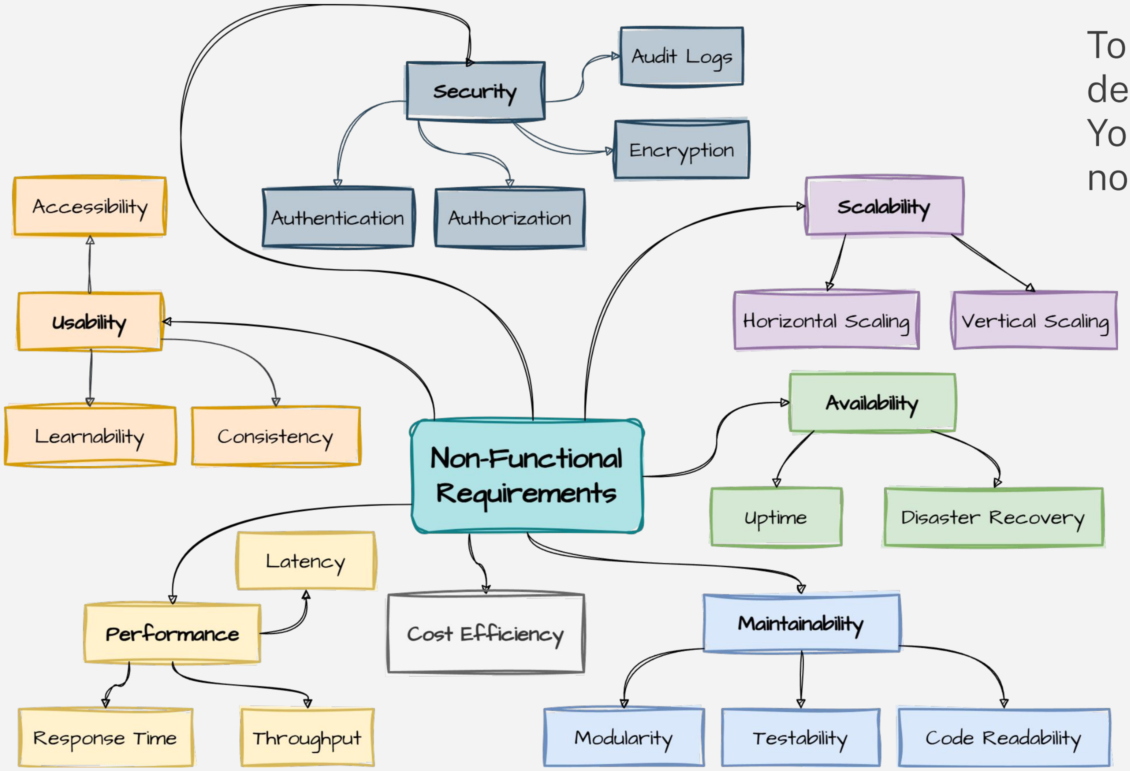
Phase 3: Observe

6. Cost optimization is incremental.
7. Unchallenged success leads to assumptions.

Phase 1: Design



Law 1: Make Cost a Non-functional Requirement



To ensure that a system is designed, developed, and operated within budget. You should consider cost as non-functional requirement (NFR)

Action:

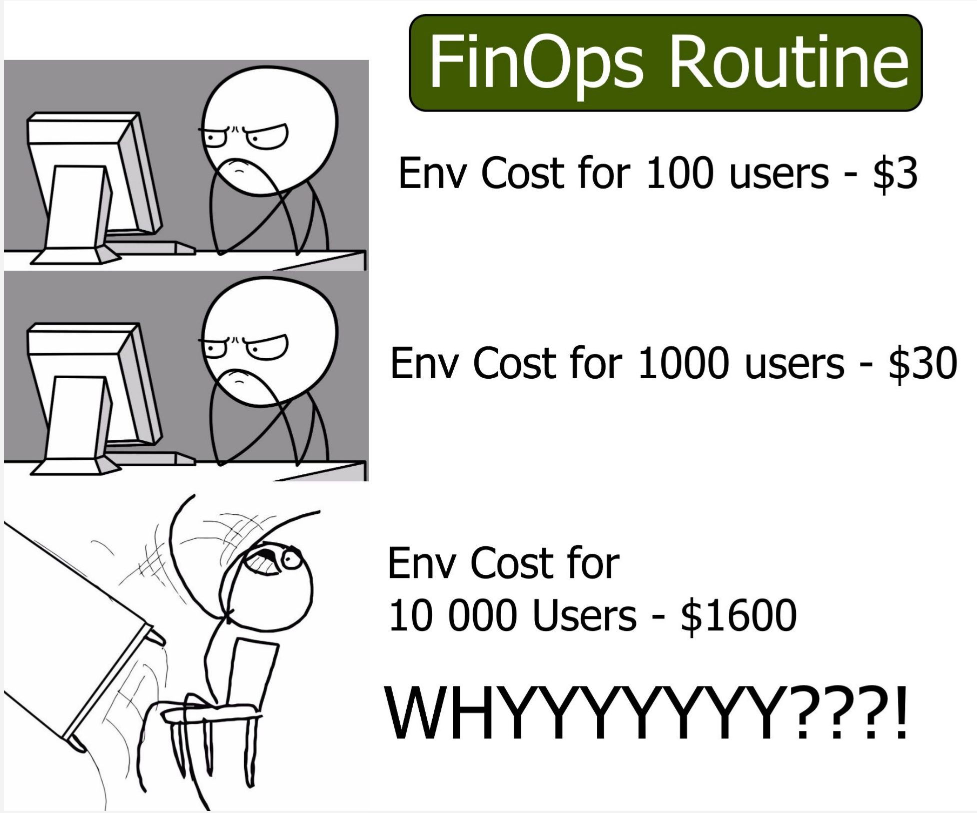
Take into account cost limitations when design the Architecture

Law 2: Systems That Last Align Cost to Business

Design systems that grow with the business and keep expenses under control to avoid problems with growth

Action:

- Be aware of how the revenue is calculated for Business.
- Your architecture documents should include cost indications.

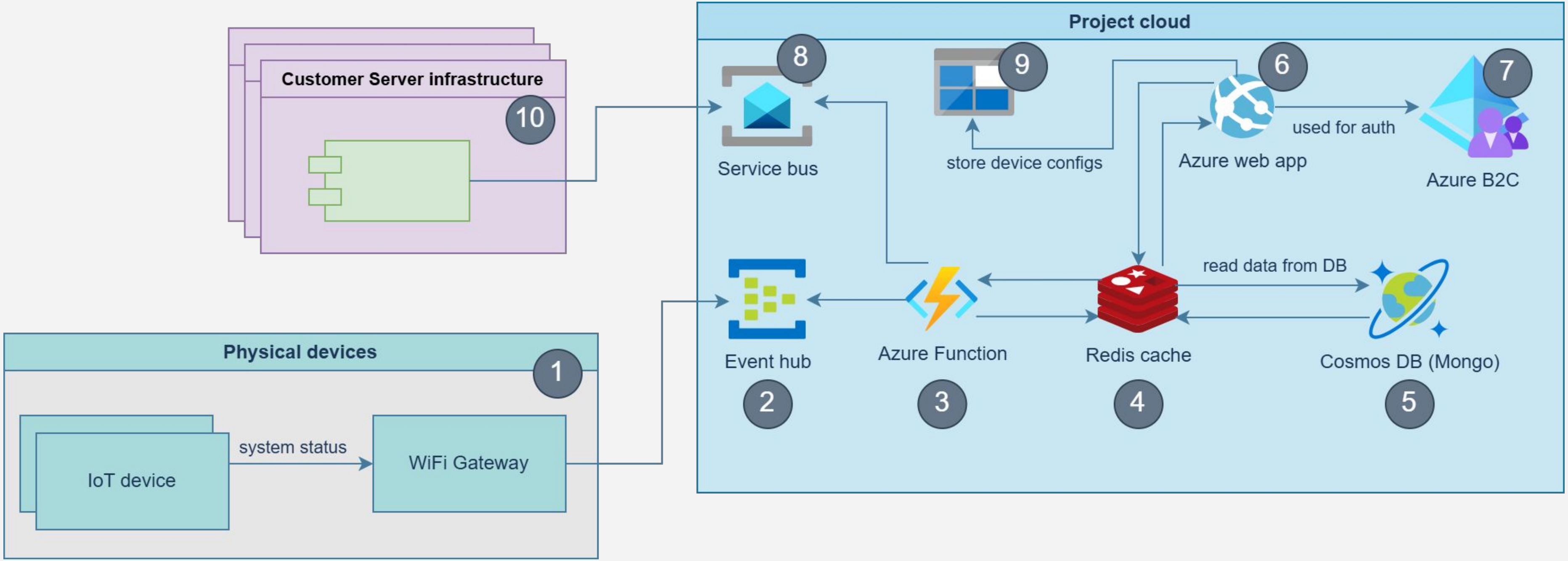


Reference:

Simplified revenue formula:

*Revenue = subscription cost -(**Infrastructure cost** / user count)*

Example: Big picture diagram documentation



#	Name of module	Description	Approximate cost for MVP per month
1	Physical devices	IoT device and Wi-Fi gateway that is used for external communication with IoT devices: Configuration, device status, etc.	Not in scope of our product
2	Event Hub	Inbound queue that is used to store device events.	Not in scope of our product
3	Azure Function	Function that subscribe on Event hub messages, then store them in DB & send to outbound queue	**$0** (consumption plan)
4	Redis Cache	Cache for store data that query frequent by system	**$16.06** (Basic C0)
5	Azure Cosmos DB	Primary database	**$25.86** (400 Request units)
6	Azure Web app	Web interface to manage gateways/devices and browse logs	**$73.00** (S1)
7	Azure B2C	Customer identity access management (CIAM)	Free (for first 50k monthly active users)
8	Azure Service bus	Outbound queue that clients use to receive device events.	**$10** (Standard tier, First 13M ops/month free)
9	Azure Blobs	Used for storing device config files	**$1**
10	Customer cloud infrastructure	External infrastructure to receive messages from IoT	Not in scope of our product
		Total	**$125.92**

Law 3: Architecting is a Series of Trade-offs

Frugality is about maximizing value, not just minimizing spend. And to do that, you need to determine what you're ready to pay for.

Action:

Include cost to your trade-offs analysis as part of Architecture Decision Record (ADR)

i receive:
Budget and Deadlines from an Alternate Reality
you receive:
A Pyramid of PowerPoint Slides That Explain Everything and Nothing
Architect

ADR Template

Status	
Context	
Decision	
Options	
Consequences	
Approximate Cost	

Include cost as part of Architecture Decision Record

ADR template examples:

https://bool.dev/l/1558

ADR 016: Outbound queue

Status	Accepted
Context	The outbound queue to send messages for the clients
Decision	Single Service bus instance for all of the regions/customers, in future switch to a Service bus instance per region when the load and amount of clients will dramatically increase
Options	<Options list>
Consequences	Positive: <list> Negative: <list>
Approximate Cost	**$10** approximate cost for MVP **$677.08** per month for a premium Service Bus instance with one unit in Post-MVP Stage. The region with a tiny load could use the standard tier (**$10**). Note: We have to pay attention some instances with the standard tier could be more expensive than Premium because standard tier has a cost **per message.**

ADR 006: Database - Option 4 Cosmos DB Mongo

Number of regions	Total data stored in transactional store	Workload mode & percentage of peak	Sample item	Multi-region writes	Price per month	Comment
1	10 GB	Variable, 30%	Item size - **1KB** Finds/sec - **1** Inserts/sec - **200** Updates/sec - **5** Deletes/sec - **1**	no	**$27.67**	Most likely situation for the MVP
1	100 GB	Variable, 30%	Item size - **1KB** Finds/sec - **2000** Inserts/sec - **2000** Updates/sec - **5** Deletes/sec - **1**	no	**$50.17**	Most likely situation for the future with 1 region.
1	1000 GB	Variable, 30%			**$275.17**	Most likely situation for the future with 1 region and increased size of storage.
4	100GB	Variable, 10%		yes	**$744.99**	Future stage with enabled multi-region writes
1	1000GB	Steady	Item size - **5 KB** Finds/sec - **2000** Inserts/sec - **2000** Updates/sec - **5** Deletes/sec - **1**	no	**$2,985.20**	The worst scenario with 1 region
4	1000GB	Steady		yes	**$12,300.05**	The worst scenario with 4 regions and multi-region writes

Price Calculators

You can calculate costs for your solutions in cloud by via following tools:

AWS

https://calculator.aws/

Azure

https://azure.microsoft.com/en-us/pricing/calculator

GCP

https://cloud.google.com/products/calculator?hl=en

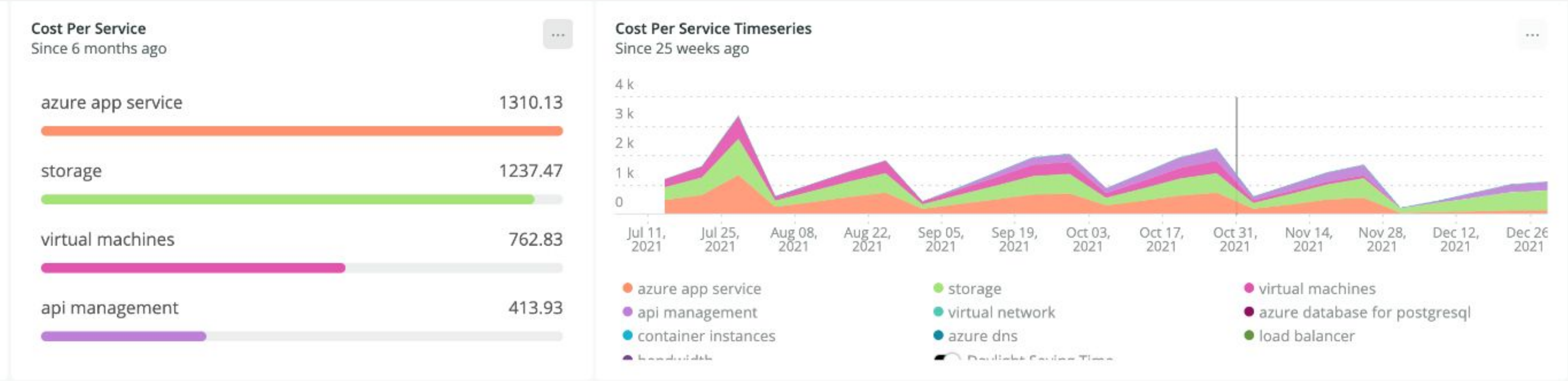
Phase 2: Measure

Law 4: Unobserved Systems Lead to Unknown Costs

If you can’t measure it, you can’t manage it.

Action:
Use tools for tracking cost and utilization of resources.

Example: NewRelic dashboard



Cost Per Service
Since 6 months ago

azure app service	1310.13
storage	1237.47
virtual machines	762.83
api management	413.93

Cost Per Service Timeseries
Since 25 weeks ago

AWS CUDOS (Cost and Usage Dashboards Operations Solution)

https://bool.dev/l/1559

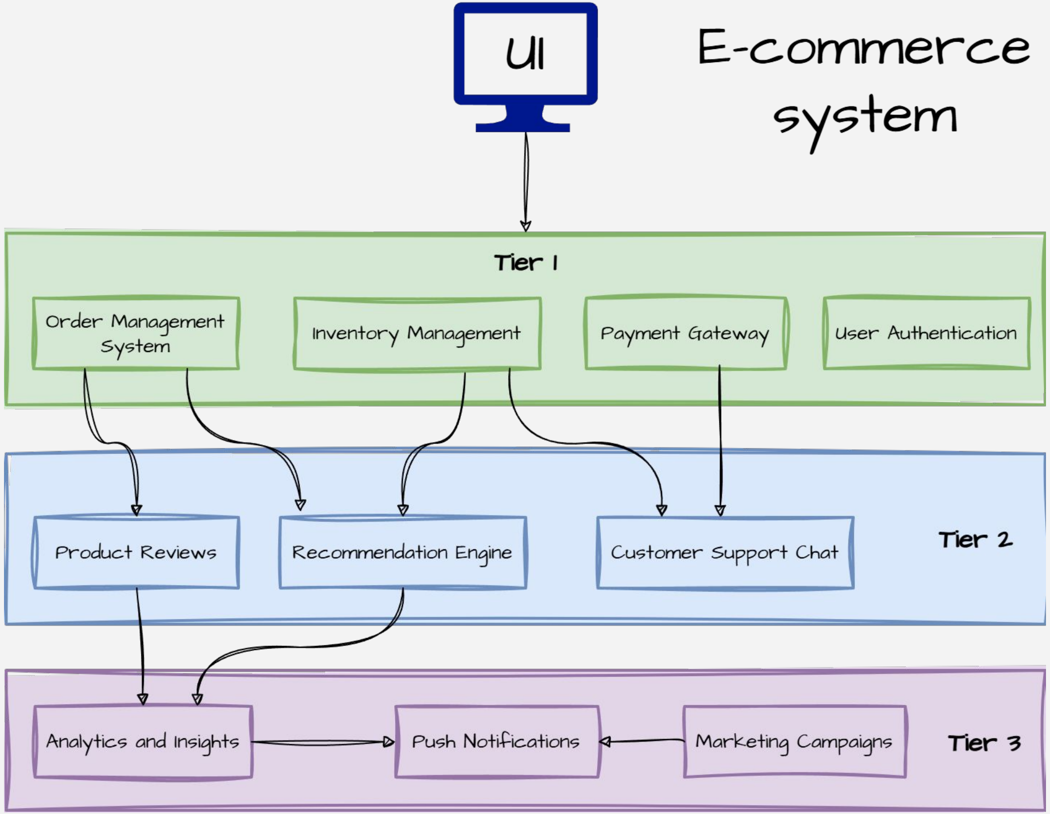
Azure: Cost Management Dashboard

https://bool.dev/l/1560

GCP: Cloud Billing Reports

https://bool.dev/l/1565

Law 5: Cost-Aware Architectures Implement Cost Controls



Evaluate your system components by criticality.

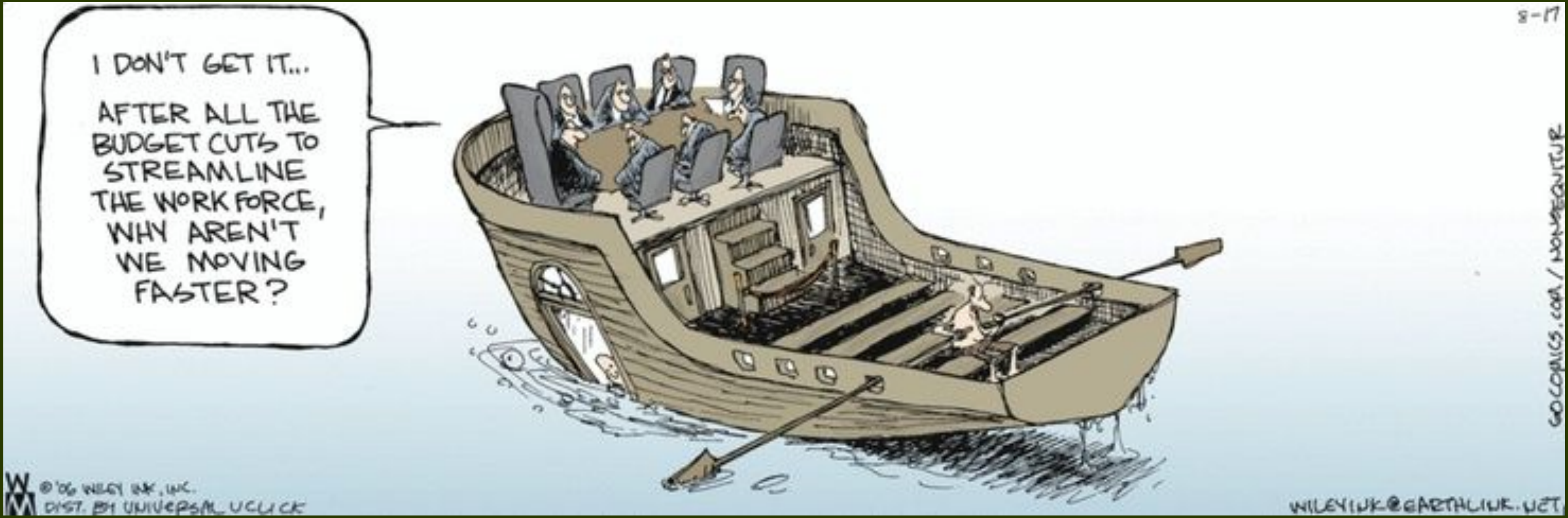
Action:

Cost optimisation must be measurable and tied (like tier 1, tier 2, tier N) to business impact.

Example: E-commerce system

- **Tier 1:** Core Components, scale regardless of cost.
- **Tier 2:** Components are important but can be temporarily scaled down without major impact.
- **Tier 3**: Components are “nice-to-have”; make them low-cost and easily controlled

Phase 3: Observe

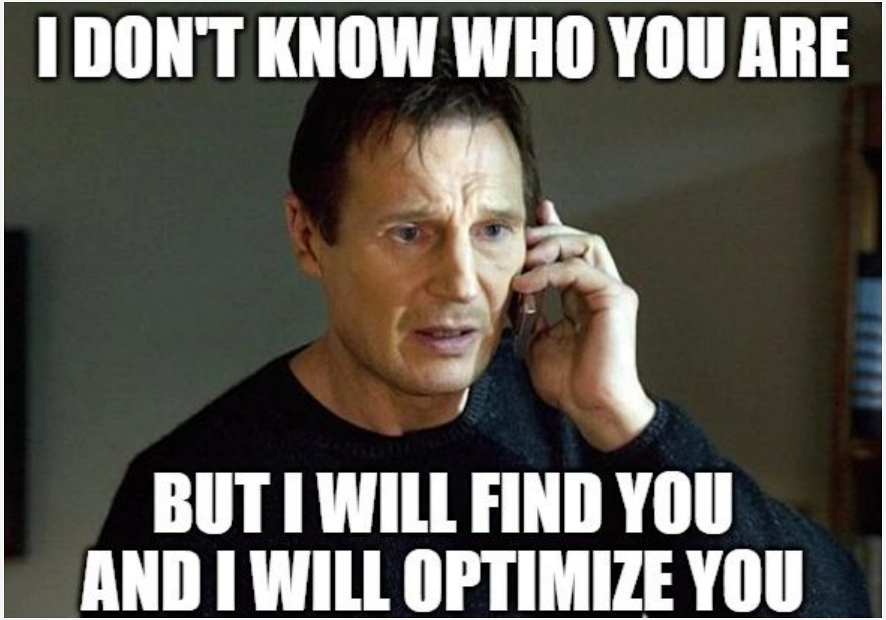

I DON'T GET IT... AFTER ALL THE BUDGET CUTS TO STREAMLINE THE WORK FORCE, WHY AREN'T WE MOVING FASTER?
8-17
GOCOMICS.COM / NONSEQUITUR
© '06 WILEY INK, INC.
DIST. BY UNIVERSAL UCLICK
WILEYINK@EARTHLINK.NET

Law 6: Cost Optimization is Incremental

Making sure your systems are cost-effective is an ongoing process. It's not something you do once and then forget about.

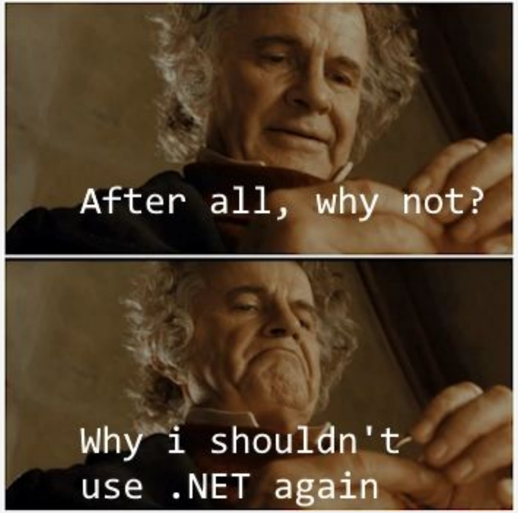
Action:

You need to keep checking your systems to find ways to make them even more efficient



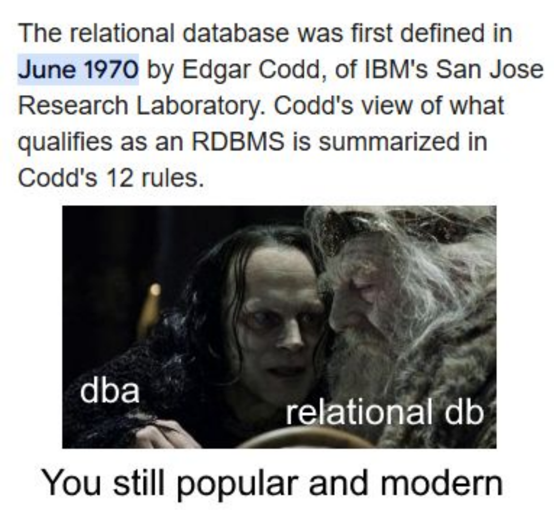
Law 7: Unchallenged Success Leads to Assumptions

- Don't Assume! Just because a solution worked in the past doesn't mean it's still the best choice today.
- Regularly challenge your assumptions and consider alternative tools and technologies that could be better suited to your current needs.



Action:

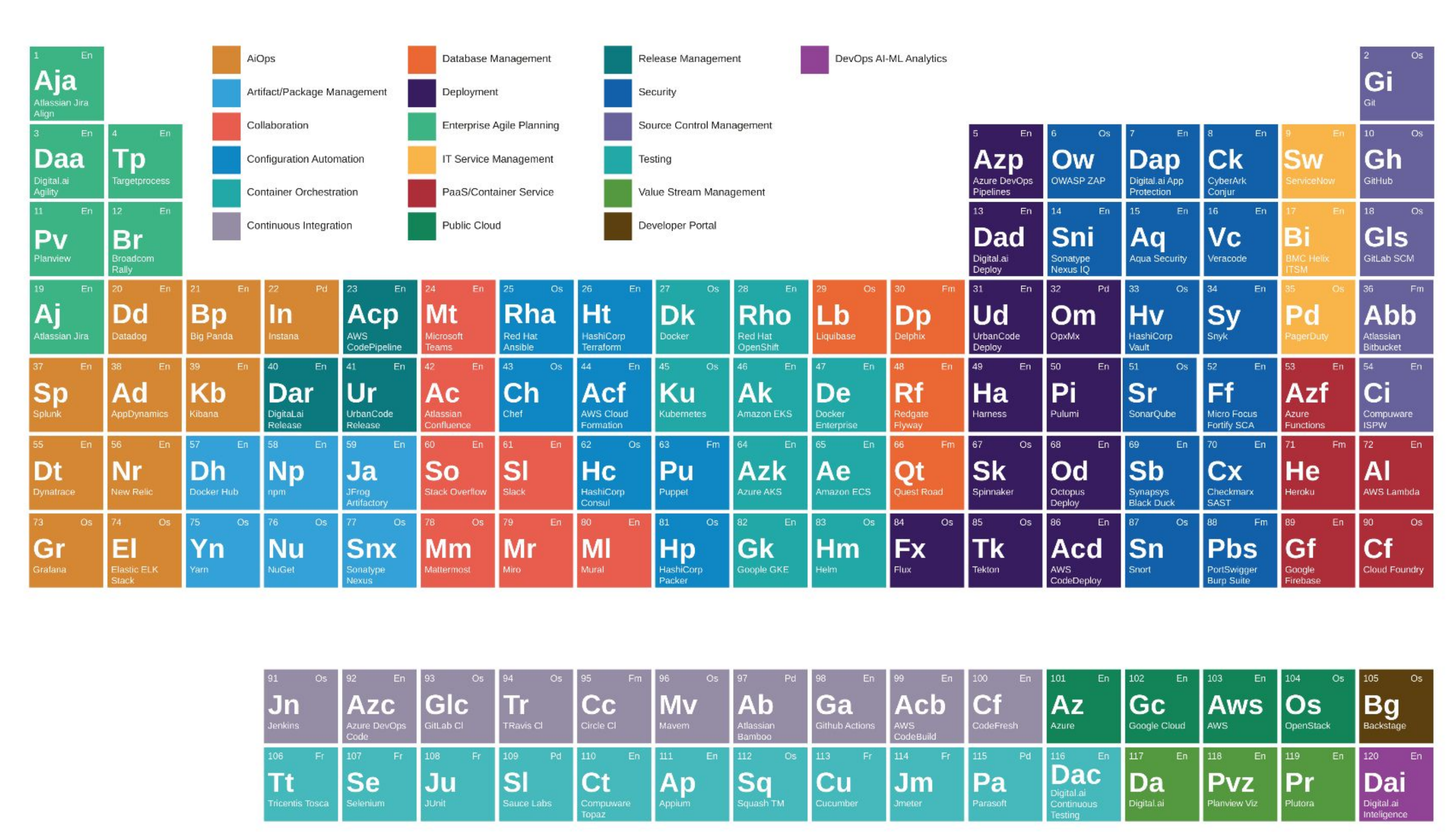
1. Periodically review the relevance and cost-effectiveness of your technologies.
2. Embrace new tools, frameworks, or cloud services that may offer better performance or lower costs.



The relational database was first defined in June 1970 by Edgar Codd, of IBM's San Jose Research Laboratory. Codd's view of what qualifies as an RDBMS is summarized in Codd's 12 rules.

DevSecOps Tools Periodic Table

https://digital.ai/learn/devsecops-periodic-table/



- AiOps
- Artifact/Package Management
- Collaboration
- Configuration Automation
- Container Orchestration
- Continuous Integration
- Database Management
- Deployment
- Enterprise Agile Planning
- IT Service Management
- PaaS/Container Service
- Public Cloud
- Release Management
- Security
- Source Control Management
- Testing
- Value Stream Management
- Developer Portal
- DevOps AI-ML Analytics

1 En Aja Atlassian Jira Align																	2 Os Gi Git
3 En Daa Digital.ai Agility	4 En Tp Targetprocess											5 En Azp Azure DevOps Pipelines	6 Os Ow OWASP ZAP	7 En Dap Digital.ai App Protection	8 En Ck CyberArk Conjur	9 En Sw ServiceNow	10 Os Gh GitHub
11 En Pv Planview	12 En Br Broadcom Rally											13 En Dad Digital.ai Deploy	14 En Sni Sonatype Nexus IQ	15 En Aq Aqua Security	16 En Vc Veracode	17 En Bi BMC Helix ITSM	18 Os Gls GitLab SCM
19 En Aj Atlassian Jira	20 En Dd Datadog	21 En Bp Big Panda	22 Pd In Instana	23 En Acp AWS CodePipeline	24 En Mt Microsoft Teams	25 Os Rha Red Hat Ansible	26 En Ht HashiCorp Terraform	27 Os Dk Docker	28 En Rho Red Hat OpenShift	29 Os Lb Liquibase	30 Fm Dp Delphix	31 En Ud UrbanCode Deploy	32 Pd Om OpxMx	33 Os Hv HashiCorp Vault	34 En Sy Snyk	35 Os Pd PagerDuty	36 Fm Abb Atlassian Bitbucket
37 En Sp Splunk	38 En Ad AppDynamics	39 En Kb Kibana	40 En Dar DigitaLai Release	41 En Ur UrbanCode Release	42 En Ac Atlassian Confluence	43 Os Ch Chef	44 En Acf AWS Cloud Formation	45 Os Ku Kubernetes	46 En Ak Amazon EKS	47 En De Docker Enterprise	48 En Rf Redgate Flyway	49 En Ha Harness	50 En Pi Pulumi	51 Os Sr SonarQube	52 En Ff Micro Focus Fortify SCA	53 En Azf Azure Functions	54 En Ci Compuware ISPW
55 En Dt Dynatrace	56 En Nr New Relic	57 En Dh Docker Hub	58 En Np npm	59 En Ja JFrog Artifactory	60 En So Stack Overflow	61 En Sl Slack	62 Os Hc HashiCorp Consul	63 Fm Pu Puppet	64 En Azk Azure AKS	65 En Ae Amazon ECS	66 Fm Qt Quest Road	67 Os Sk Spinnaker	68 En Od Octopus Deploy	69 En Sb Synapsys Black Duck	70 En Cx Checkmarx SAST	71 Fm He Heroku	72 En Al AWS Lambda
73 Os Gr Grafana	74 Os El Elastic ELK Stack	75 Os Yn Yarn	76 Os Nu NuGet	77 Os Snx Sonatype Nexus	78 Os Mm Mattermost	79 En Mr Miro	80 En Ml Mural	81 Os Hp HashiCorp Packer	82 En Gk Google GKE	83 Os Hm Helm	84 Os Fx Flux	85 Os Tk Tekton	86 En Acd AWS CodeDeploy	87 Os Sn Snort	88 Fm Pbs PortSwigger Burp Suite	89 En Gf Google Firebase	90 Os Cf Cloud Foundry
			91 Os Jn Jenkins	92 En Azc Azure DevOps Code	93 Os Glc GitLab CI	94 Os Tr TRavis CI	95 Fm Cc Circle CI	96 Os Mv Mavem	97 Pd Ab Atlassian Bamboo	98 En Ga Github Actions	99 En Acb AWS CodeBuild	100 En Cf CodeFresh	101 En Az Azure	102 En Gc Google Cloud	103 En Aws AWS	104 Os Os OpenStack	105 Os Bg Backstage
			106 Fr Tt Tricentis Tosca	107 Fr Se Selenium	108 Fr Ju JUnit	109 Pd Sl Sauce Labs	110 En Ct Compuware Topaz	111 En Ap Appium	112 Os Sq Squash TM	113 Fr Cu Cucumber	114 Fr Jm Jmeter	115 Pd Pa Parasoft	116 En Dac Digital.ai Continuous Testing	117 En Da Digital.ai	118 En Pvz Planview Viz	119 En Pr Plutora	120 En Dai Digital.ai Inteligence

Common Pitfalls

Pitfall: Ignoring Database Growth

Over time, databases grow unchecked with unused or unnecessary data, leading to increased storage costs, slower performance, and higher query execution times.

How to avoid:

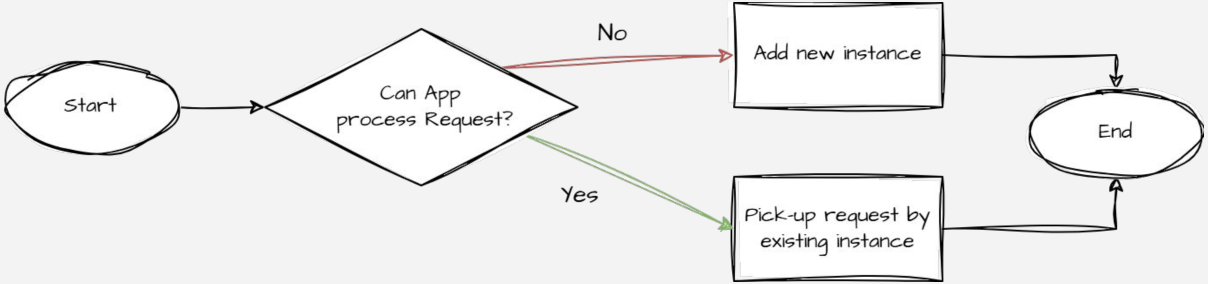
- Move old unused data to cheaper storage solutions (e.g., cold storage).
- Regularly evaluate schema designs, optimize indexes, and normalize or denormalize where it aligns with access patterns. Ensure data size and types are right-sized.
- Continuously track database size and growth trends to plan scaling and cost optimization.

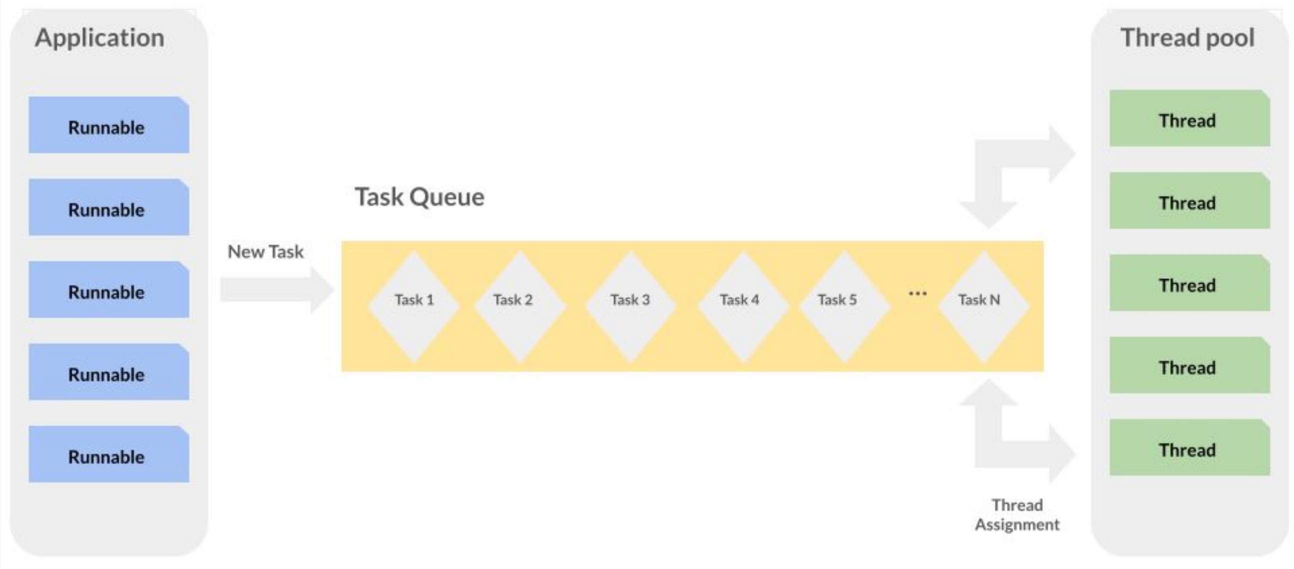
Pitfall: Inefficient Use of IO-Bound Operations

Relying on blocking IO operations can reduce available threads in the thread pool, causing performance bottlenecks and forcing scaling to handle the load.

How to avoid:

Use non-blocking or asynchronous IO mechanisms (e.g., event loops in Node.js, async/await in modern languages).



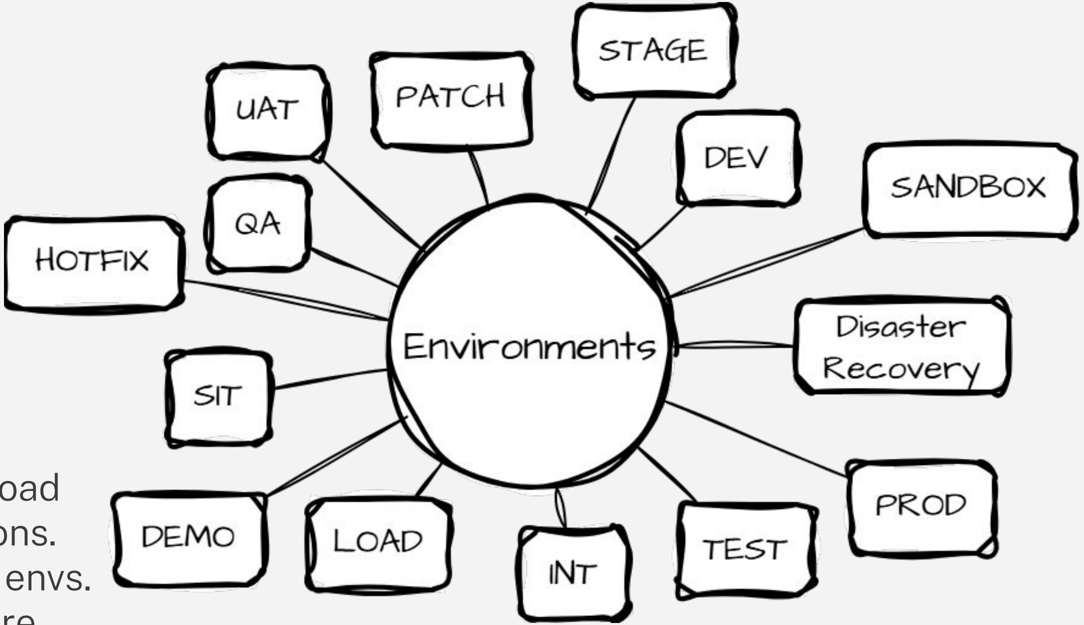


Pitfall: Over-Provisioning Resources

1. Provisioning more compute, storage, or RAM than necessary wastes money without delivering proportional benefits.
2. Create unnecessary environments, increasing infrastructure cost.

How to avoid:

1. Implement right-sizing strategies and auto-scaling policies. Review resource utilization metrics frequently and apply load testing to determine optimal configurations.
2. Review env policy and keep only needed envs. Combine env's on same service plan where possible (like dev/QA envs)



Pitfall: Lack of Team Ownership

Teams without clear ownership of cost and performance often make decisions that might be not optimal.

How to avoid:

- Each Team member should be aware how much we pay for infrastructure.
- Apply FinOps principles to make cost-efficiency an ongoing focus.

Principles

- Teams need to collaborate
- Decisions are driven by business value of cloud
- Everyone takes ownership for their cloud usage
- FinOps data should be accessible and timely
- A centralized team drives FinOps
- Take advantage of the variable cost model of the cloud

Domains & Capabilities

FinOps Foundation

Understand Cloud Usage & Cost

- Data Ingestion
- Allocation
- Reporting & Analytics
- Anomaly Management

Quantify Business Value

- Planning & Estimating
- Forecasting
- Budgeting
- Benchmarking
- Unit Economics

Optimize Cloud Usage & Cost

- Architecting for Cloud
- Workload Optimization
- Licensing & SaaS
- Rate Optimization
- Cloud Sustainability

Manage the FinOps Practice

- FinOps Practice Operations
- FinOps Education & Enablement
- Cloud Policy & Governance
- Invoicing & Chargeback
- FinOps Assessment
- Onboarding Workloads
- FinOps Tools & Services
- Intersecting Disciplines

Core Personas

- Engineering
- FinOps Practitioner
- Finance
- Leadership
- Procurement
- Product

Allied Personas

- ITAM
- ITFM
- ITSM
- Security
- Sustainability

Maturity

- Crawl
- Walk
- Run

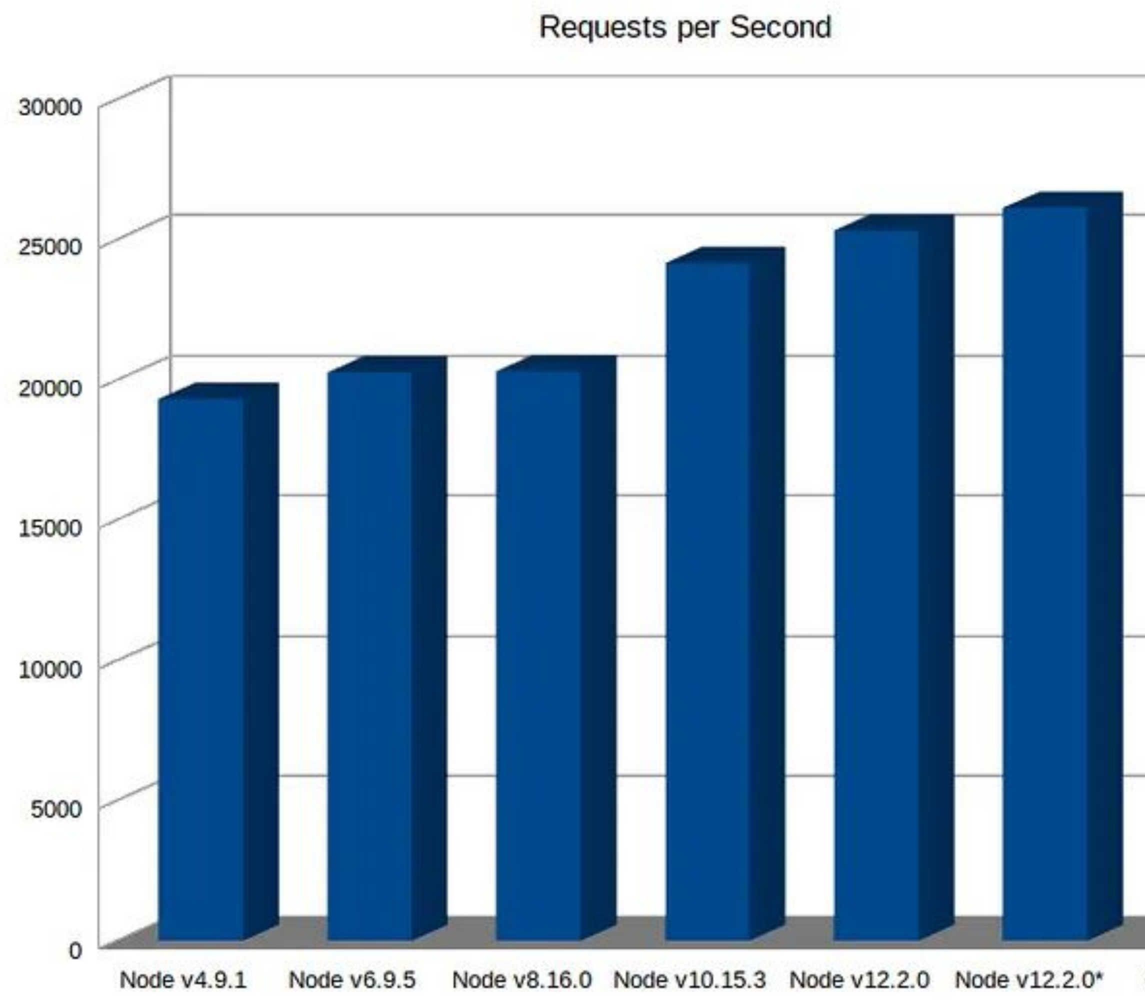
https://www.finops.org/framework

Pitfall: Ignoring Technical Debt

Allowing technical debt to accumulate reduces agility and increases costs to implement future changes.

How to avoid:

Dedicate time in sprints to reduce technical debt. Use static code analysis and architectural reviews to identify and address problematic areas early.



Pitfall: Over-Engineering Solutions

Using complex architectures (e.g., microservices for small-scale apps) adds unnecessary overhead in terms of development, maintenance, and runtime costs.

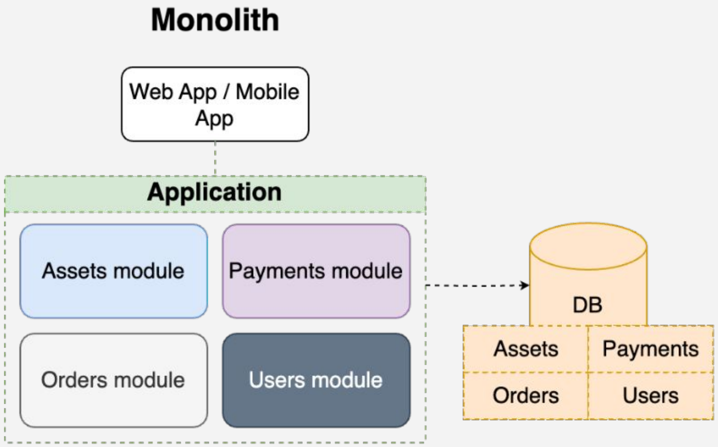
we have to create service skeleton, lets apply clean arch, CQS and DDD approach
and UoW
we have to build common frmk
can we create a service seleton then add complexity when required
service skeleton creation task couldnt take less then a few days of work
Fired: reason do not use best OOP practice and clean architecture design

How to avoid:

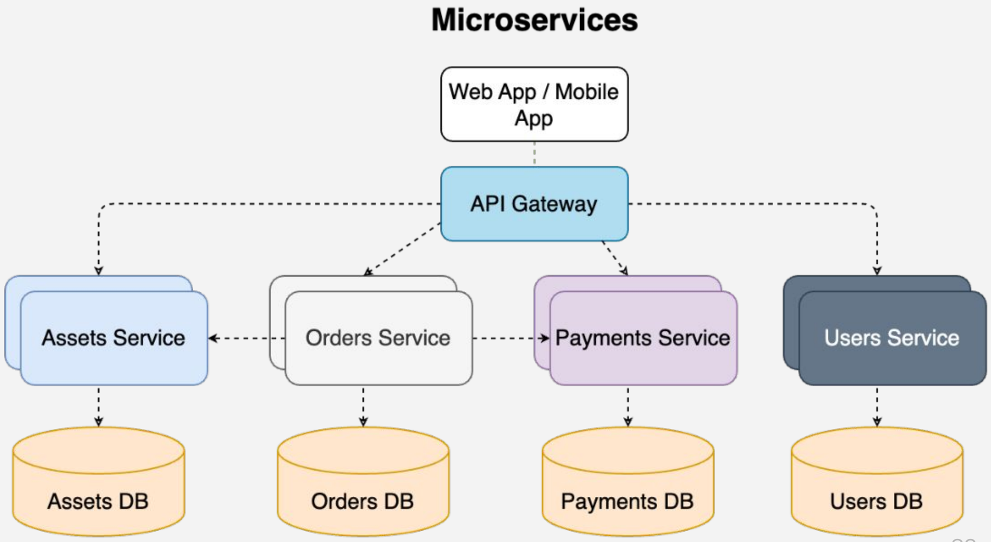
Start simple with monoliths or modular monoliths, scaling into distributed systems when reach by specific scaling or team requirements.

DDD approach, Vertical Slice Architecture could be a good choice for monolith to preparing the system for future splitting.

Example: Is Monolith Frugal?



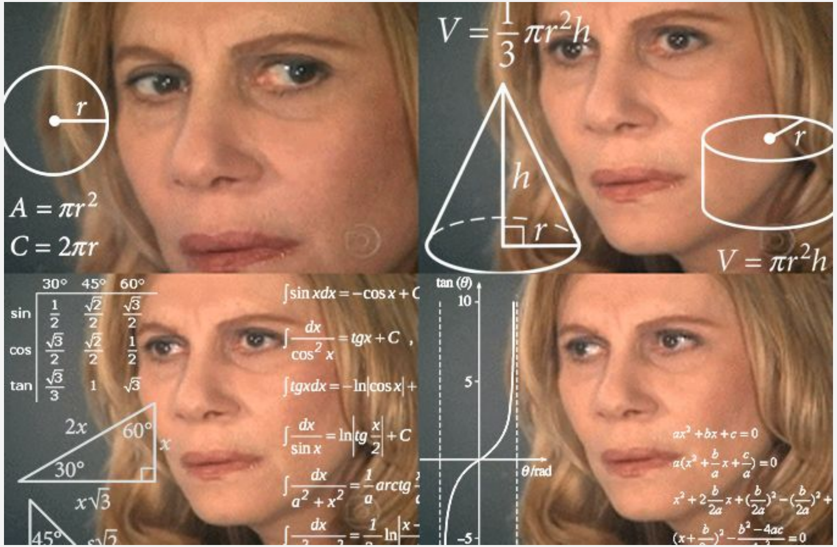
VS



Example: Is Monolith Frugal - Cost calculation

Baseline

- Use Azure App service from West Europe with Windows Operating system
- Assume that App Service **S1** (1 core, 1.75 GB RAM, and 50 GB store) has throughput **1 000** requests
- Assume that App Service **S3** (4 cores, 7gb RAM, and 50 GB store) has throughput **4 000** requests
- Database & rest out of scope for current estimation



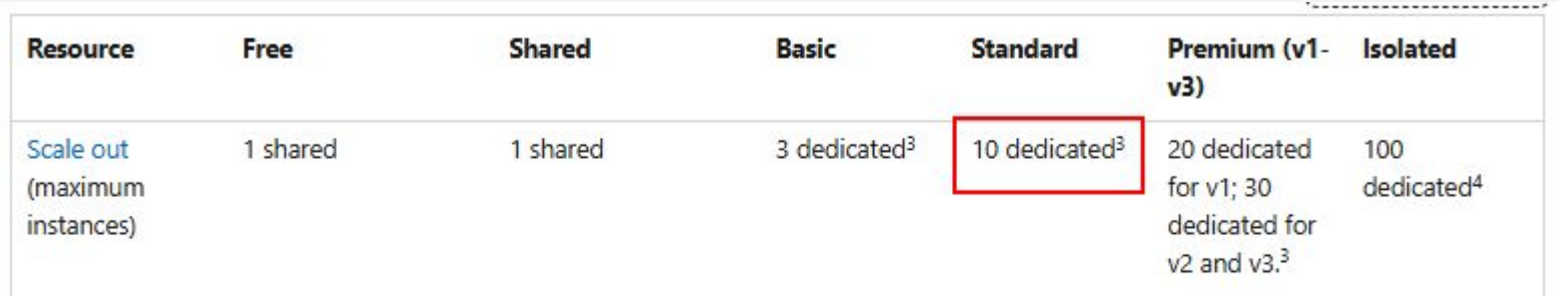
Let's Calculate!

App service plan	OS	Price per month	Used for
S1 (1 core, 1.75 GB ram and 50 GB store)	Windows	**$73**	Microservice
S3 (4 cores, 7gb ram, 50 GB store)	Windows	**$292**	Monolith

Example: Is Monolith Frugal - Ramp up

Scenario		Monolith		Microservices		
Load Scenario	**Total Requests count**	**Instances (App Service S3)**	**Cost per month**	**Total Instances (Breakdown)**	**Instances (App Service S1)**	**Cost per month**
Baseline: 1k requests per module	4 000	1	$292	Assets: **1**; Payments: **1**; Orders: **1**; Users: **1**	4	$292
+ 4k requests for Assets	8 000	2	$584	Assets: **5**; Payments: **1**; Orders: **1**; Users: **1**	8	$584
+ 10k requests for Payments	18 000	5	$1460	Assets: **5**; Payments: **11**; Orders: **1**; Users: **1**	18	$1314
+ 80k requests for orders	98 000	25	$7300	Assets: **5**; Payments: **11**; Orders: **81**; Users: **1**	98	$7154

Example: Is Monolith Frugal - Limitations



Resource	Free	Shared	Basic	Standard	Premium (v1-v3)	Isolated
Scale out (maximum instances)	1 shared	1 shared	3 dedicated[3]	10 dedicated[3]	20 dedicated for v1; 30 dedicated for v2 and v3.[3]	100 dedicated[4]

- To scale beyond **10** instances, you should move to a **Premium** / **Isolated** plan, which significantly increases costs for monolith.
- With microservices, individual services can be scaled **independently**, helping you reduce costs by only scaling the most critical services (e.g., consider premium for Orders service and Standard for rest).

Example: Is Monolith Frugal - Final Thoughts

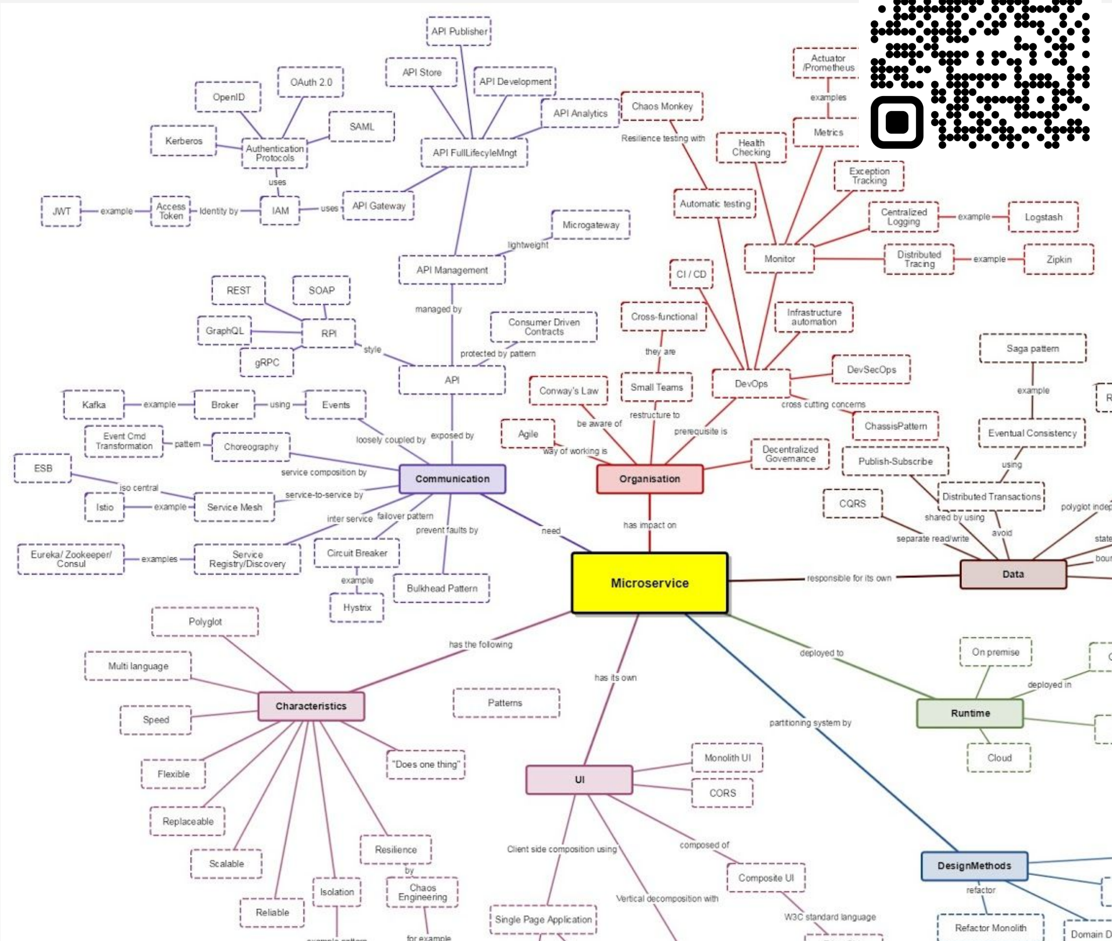
High res image

Hidden Extra Costs with Microservices:

- Infrastructure: API Gateway, Message Broker, and other components.
- Development: Higher complexity and need for skilled engineers.
- Increased DevOps effort and operational costs.

Best Fit:

- Microservices excel in complex or growing systems.
- For simpler applications, monolithic architectures may offer better cost efficiency due to reduced complexity.



Key takeaways

Find rich customer that don't care about infrastructure cost

Key takeaways

- Align architecture with business needs and technical constraints.
- Make cost a non-functional requirement and consider as part of trade-off analysis
- Monitor and optimize cost continuously
- Design systems to scale efficiently and avoid unnecessary spending.
- Avoid pitfalls that might increase your infrastructure costs.

Q&A

Thank you!