

DACODES · FULL GUIDE

Monolith vs Microservices

The full breakdown: how to choose, when to split, and the templates to do it without breaking everything.



BEFORE YOU START

A guide for teams deciding how their system **should grow**

WHAT YOU GET

The decision framework from the video, explained properly, plus 7 fill-in templates your team can use this week.

WHO IT IS FOR

CTOs, tech leads, founders and product people deciding how their system should grow.

TIME TO READ

15 minutes. Time to use: one workshop with your team.

HOW TO USE THIS GUIDE

Read Levels 1 to 6 once. Then run the Readiness Scorecard (Template 1) with your team.

Your score tells you which other templates you need. Most teams only need two or three of them.

CONTENTS

What's inside

–	The 60-second version	04
L1	The monolith	06
L2	Where it hurts	08
L3	Microservices, done right	11
L4	The bill nobody puts on the slide	14
L5	How to choose	16
L6	The AI angle	20
–	The 5 most expensive mistakes	22
T	Templates (7)	23
–	Glossary	33

START HERE

The 60-second version

Every software team hits the same fork sooner or later: one big codebase, or dozens of small services? Here is the whole answer in four lines.

01

Monolith

Fastest way to build while you are small. One codebase, one deploy, one database.

02

Microservices

Let you scale and ship parts independently. You pay for it in operations.

03

The default

Start with a modular monolith: one deploy, clear internal boundaries.

04

When to split

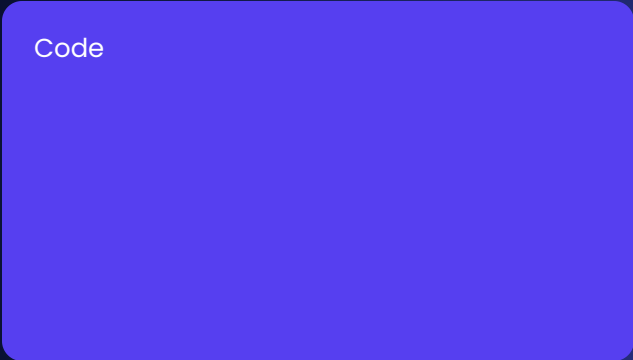
Only when the pain is real and measurable. One service at a time.

THE ONE IDEA TO REMEMBER

Microservices are not a badge of maturity. They are a **trade-off**.

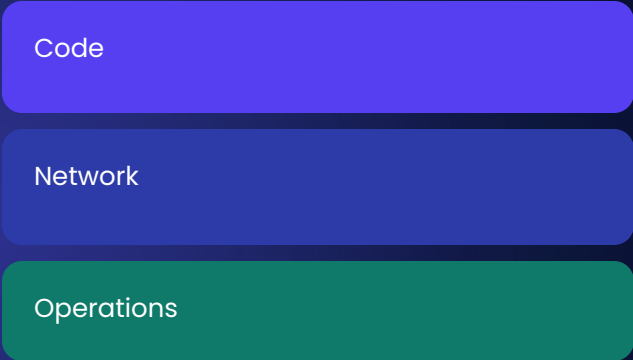
Complexity does not disappear when you split a system; it moves from your code to your network and your operations.

MONOLITH

A large, single purple rounded rectangle representing a monolith.

Code

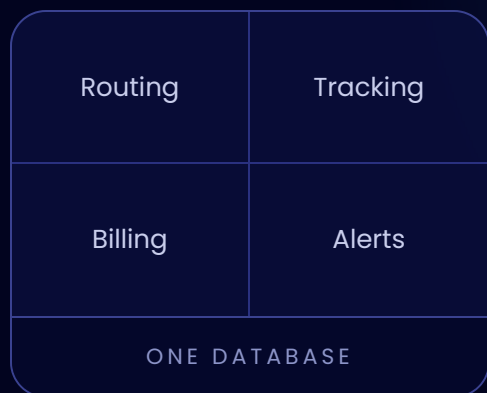
MICROSERVICES

Three stacked rounded rectangles representing microservices: a purple one for Code, a blue one for Network, and a green one for Operations.

Code

Network

Operations



RUTAVO AS A MONOLITH

01

LEVEL 1

The monolith

A monolith is one application that contains every part of your product. Throughout this guide we use the same example as the video: Rutavo, a logistics startup with four parts. Routing, Tracking, Billing and Alerts. In a monolith, all four live in one codebase, ship in one deploy and share one database.

LEVEL 1

Why it is a **gift** early on

- **One repo, one build, one debugger.** A new developer can run the whole product on their laptop on day one.
- **Easy tests.** End-to-end tests run in one process, with no network in the middle.
- **Simple data.** One database means one transaction can update an order and its invoice together. No sync problems.
- **Cheap to run.** One thing to host, monitor and keep alive.
- **Fast to change direction.** When you are still finding product-market fit, moving a feature means moving code, not renegotiating APIs between teams.

A MONOLITH IS NOT LEGACY

Plenty of large, successful products run on well-structured monoliths. The problem is never “monolith”. The problem is a monolith with no internal boundaries, where everything can touch everything.

02

LEVEL 2

Where it hurts

As the product and the team grow, the same simplicity starts to cost you. These are the four pains that show up first.

LEVEL 2

The four pains

PAIN

WHAT IT LOOKS LIKE DAY TO DAY

Every change ships everything

A one-line fix in Alerts means redeploying the entire app. A bad deploy can take the whole platform down.

Teams collide

Several teams in one codebase step on each other. Merge conflicts, release trains, and every change becomes a negotiation.

You scale everything to scale one part

Tracking gets 50x more traffic than Billing, but you can only add more copies of the whole app. You pay for capacity you do not need.

Slow feedback

Builds and test suites take longer every month. Developers wait more and ship less.

TRAFFIC, RUTAVO EXAMPLE

Tracking



50x

Billing



1x

SELF-CHECK

Warning signs checklist

Tick every statement that is true for your team today. Three or more ticks means it is time to read Level 5 carefully.

-
- ☐ Deploys need coordination between more than two teams.
 - ☐ We have delayed a release because of a change in a part of the code we did not touch.
 - ☐ One part of the system needs very different scaling than the rest.
 - ☐ A failure in a non-critical feature has taken down a critical one.
 - ☐ The full build and test cycle takes more than 30 minutes.
 - ☐ New developers take more than a month to ship confidently.
 - ☐ Nobody can say clearly who owns certain parts of the code.
 - ☐ We avoid touching some modules because “something always breaks”.
-

Routing

OWN DATA

Tracking

OWN DATA

Billing

OWN DATA

Alerts

OWN DATA

RUTAVO AS SERVICES

03

LEVEL 3

Microservices, done right

Microservices split the product into small services that each do one job. The key word is business capability: you split by what the business does, not by technical layer. Rutavo becomes a Routing service, a Tracking service, a Billing service and an Alerts service.

LEVEL 3

The three rules of a **real service**

RULE 1**Own data**

Each service has its own database. Nobody else reads or writes it directly.

If two services share tables, they are still one system. You just made it slower.

RULE 2**Clear API**

Other services talk to it only through a documented, versioned contract.

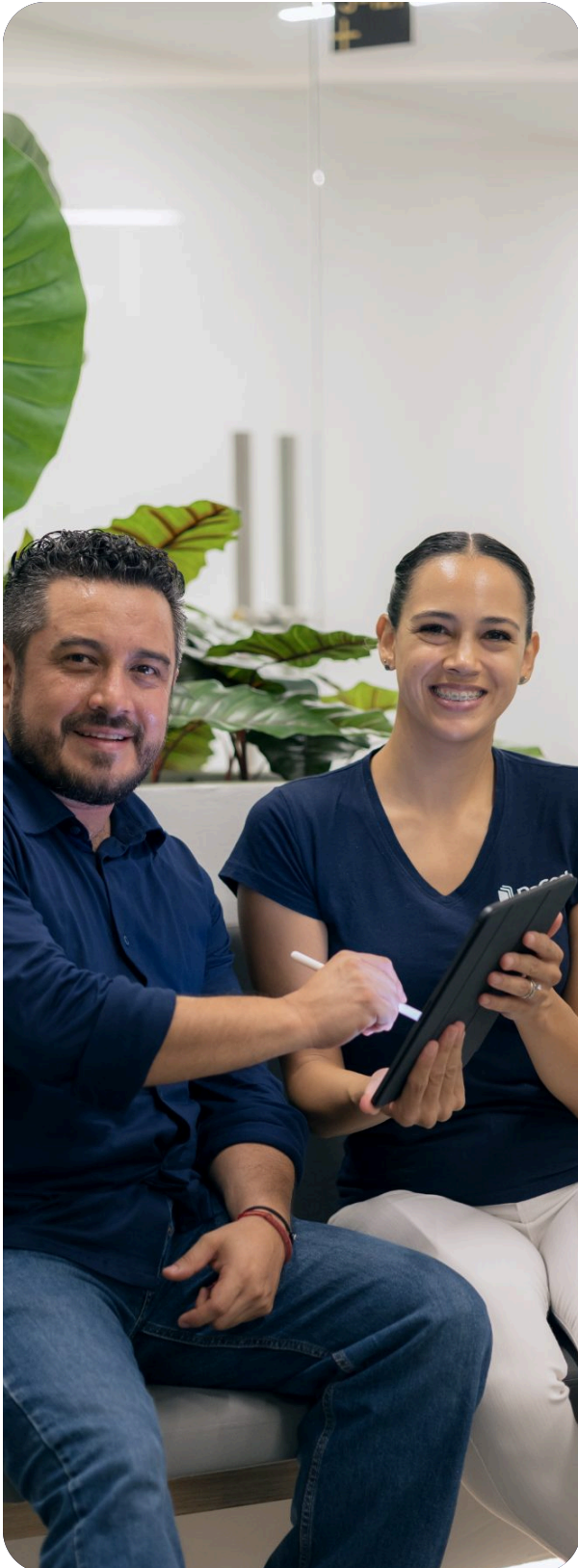
Teams can change internals freely without breaking each other.

RULE 3**Own schedule**

The team ships it whenever it is ready, on its own version.

Billing can deploy ten times a day without touching Tracking.

Top: what it means. Bottom: why it matters.



LEVEL 3

What you get **in return**

- **Deploy independently:** smaller, safer releases.
- **Fail independently:** if Alerts goes down, customers can still track their delivery.
- **Scale independently:** add capacity only where the traffic is.
- **Clear ownership:** one team, one service, one pager.

04

LEVEL 4

The bill nobody puts on the slide

That freedom comes with a bill. Before you split anything, make sure you can pay each of these line items.

RULE OF THUMB

If you do not yet have automated deploys, centralised logs and basic monitoring for one app, you are not ready to run ten.

LEVEL 4

Seven line items, and **why** you suddenly need them

01**Retries and timeouts**

Networks fail. A function call that never failed now can, so every call needs a plan for when it does.

03**Data consistency**

Data that used to live in one transaction now lives in several databases. “Eventually consistent” is real, and your product has to handle it.

05**Platform and DevOps**

CI/CD per service, container orchestration, service discovery, secrets, environments. Someone has to build and run all of it.

07**On-call load**

Every service needs an owner who answers when it breaks at 3 a.m.

02**Circuit breakers**

When one service is slow, callers must stop hammering it or the slowness spreads to everything.

04**Distributed tracing**

One user request may hop through six services. Without tracing, finding the slow or broken one is guesswork.

06**Cloud cost**

More services means more instances, more network traffic and more monitoring. Your cloud bill grows with every service.

05

LEVEL 5

How to choose

step 1	Start with a modular monolith	17
step 2	Split only for a real pain	17
step 3	Peel off one service at a time	18

STEP 1

Start with a modular monolith

One codebase and one deploy, but with clear modules inside. Each module owns its own tables and exposes a small internal interface. Other modules never reach into its internals. You keep the speed of a monolith and you prepare the seams you will need if you ever split.

STEP 2

Split only for a real pain

There are three good reasons to extract a service. If you cannot point to one of them with data, do not split.

REAL PAIN

HOW TO PROVE IT

It needs to scale alone

Traffic or resource use for one module is clearly different from the rest, and scaling the whole app is costing you money or performance.

A team is blocked

A team regularly waits on other teams to release, and the delay shows up in your delivery metrics.

Deploys are risky

Changes in one area keep breaking unrelated areas, or one fragile part forces slow, careful releases for everyone.

STEP 3

Peel off **one service at a time**

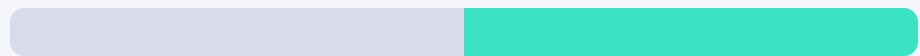
Pick the module with the clearest pain and the cleanest boundary. Extract it, run it in production, learn, then decide on the next one. This is often called the **strangler pattern**: new traffic is routed to the new service while the old code is gradually retired.

THE STRANGLER PATTERN

Route a small share



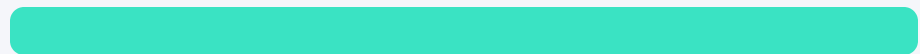
Increase gradually



Service is stable



Old path removed



■ Old code path ■ New service

- 01 Put the module behind a clear internal interface inside the monolith.
- 02 Give it its own data. Migrate its tables and stop other modules from reading them directly.
- 03 Stand up the new service with the same interface, plus monitoring and tracing from day one.
- 04 Route a small share of traffic to it. Compare results. Increase gradually.
- 05 Remove the old code path once the service is stable. Then review before extracting the next one.

DECIDE FOR YOUR ROADMAP, NOT
FOR THE TREND

The right
architecture is the
one that lets your
team ship what is
on the roadmap for
the next 12 to 18
months.

Not the one that looks best in a
conference talk.

06

LEVEL 6

The AI angle

AI agents are becoming new “users” of your system. An agent that books deliveries, answers customers or reconciles invoices needs to call your software, just like another service would. Whether you run one service or fifty, agents need clean, safe APIs.

LEVEL 6

What makes a system **agent-ready**

- **Well-defined actions.** Small, clearly named endpoints (for example “create shipment”, “get delivery status”) are much easier for an agent to use correctly than one giant endpoint with twenty options.
- **Clear contracts.** Typed inputs and outputs, documented errors and examples. Agents read your docs literally.
- **Safe permissions.** Agents get their own credentials with the minimum access they need. Read and write are separated.
- **Guardrails on risky actions.** Refunds, deletions or payments need limits, confirmation steps or human approval.
- **Traceability.** Every action an agent takes is logged and attributable, so you can audit and undo.

THE GOOD NEWS

The work that makes a monolith modular (clear boundaries, clean interfaces, owned data) is the same work that makes it safe for agents. You do not need microservices to be AI-ready. You need good seams.

WATCH OUT

The 5 most **expensive mistakes**

-
- 01 **The distributed monolith.** Services that must all be deployed together, or that share one database. You get all the cost of microservices and none of the benefits.

 - 02 **Splitting by technical layer.** A “frontend service”, a “logic service” and a “database service” is not microservices. Split by business capability.

 - 03 **Too many, too fast.** Going from one app to twenty services in a quarter. Extract one, learn, then the next.

 - 04 **Skipping the platform work.** No tracing, no central logs, no automated deploys. Debugging becomes archaeology.

 - 05 **Splitting to fix bad code.** If the code is tangled inside the monolith, it will be tangled across the network too. Fix the boundaries first.
-

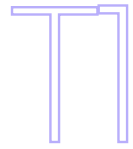
TOOLKIT

Templates

Copy these into your own docs or print them for a workshop. Each one has a short example row so you can see how to fill it in.

01	Architecture Readiness Scorecard	You want an honest answer to "should we split?"
02	Pain Log	You need evidence before making the call.
03	Module Boundary Map	You are making your monolith modular.
04	Service Extraction Brief	You have picked the first service to peel off.
05	Operational Readiness Checklist	Before a new service goes to production.
06	Agent-Ready API Checklist	Before an AI agent touches your system.
07	Architecture Decision Record	Whenever you make the call, so future you knows why.

TEMPLATE 1



Architecture Readiness Scorecard

Score each statement from 1 (not true at all) to 5 (completely true). Do it as a team and use the median, not the loudest voice.

#	STATEMENT	SCORE (1 TO 5)
1	Different parts of our product have very different traffic or scaling needs.	<input type="text"/>
2	We have three or more teams working in the same codebase.	<input type="text"/>
3	Teams regularly wait on each other to release.	<input type="text"/>
4	A bug in one area regularly breaks unrelated areas.	<input type="text"/>
5	Our product and domain are stable; we are past finding product-market fit.	<input type="text"/>
6	We already have clear module boundaries inside our code.	<input type="text"/>
7	We have automated deploys (CI/CD) we trust.	<input type="text"/>
8	We have centralised logs, monitoring and alerting.	<input type="text"/>
9	We have people with real experience running distributed systems.	<input type="text"/>
10	We have budget for higher infrastructure and operations costs.	<input type="text"/>
TOTAL		<input type="text"/>

TEMPLATE 1 · RESULTS

What your **score** means

10 to 22

Stay with a monolith

23 to 36

Modular monolith

37 to 50

Selective microservices

SCORE

RECOMMENDATION

10 to 22

Stay with a monolith. Invest in modular boundaries (Template 3) and in your delivery pipeline. Revisit in 6 months.

23 to 36

Modular monolith, with one extraction if the Pain Log (Template 2) shows a clear winner. Fill in Template 4 for that one service only.

37 to 50

You have a strong case for selective microservices. Prioritise extractions by pain, and run Template 5 before each one goes live.

READ THE SPLIT, NOT JUST THE TOTAL

Statements 1 to 5 measure how much you need to split. Statements 6 to 10 measure how ready you are to pay the bill. High need with low readiness means: fix the platform first.

Pain Log

T2

[illegible]

DACODES · FULL GUIDE

TEMPLATE 3

T3

Module **Boundary Map**

List each business capability in your product. This becomes the blueprint for a modular monolith, and later for any service you extract.

MODULE	WHAT IT DOES	OWNER TEAM	DATA IT OWNS	DEPENDS ON	CHANGE RATE
<i>Tracking</i>	<i>Live location and delivery status for each shipment</i>	<i>Ops squad</i>	<i>shipments_status, gps_events</i>	<i>Routing</i>	<i>Weekly</i>

TEMPLATE 3

Boundary health check

- ☐ Every module has exactly one owner team.
 - ☐ Every table belongs to exactly one module.
 - ☐ Modules only call each other through a defined interface, never directly into internals.
 - ☐ No circular dependencies (A depends on B and B depends on A).
 - ☐ Each module can be described in one sentence a non-engineer understands.
-

TEMPLATE 4



Service Extraction Brief

One page per service you plan to peel off. If you cannot fill in every row, you are not ready to extract it yet.

FIELD

YOUR ANSWER

Service name

Business capability (one sentence)

The real pain it solves (link to Pain Log)

How we will measure success

Owner team and on-call

API contract (main endpoints or events)

Data it will own

Data migration plan

Services and modules that call it

Failure plan (what callers do if it is down)

Rollout plan (traffic percentages, dates)

Rollback plan

Estimated extra monthly infra cost

Go / No-go decision and date

TEMPLATE 5

T5

Operational Readiness Checklist

Every box ticked before the new service takes real traffic.

Deploy

- ☐ Automated build, test and deploy pipeline of its own.
- ☐ Can deploy without deploying anything else.
- ☐ One-command rollback, tested at least once.

Observe

- ☐ Logs go to the central logging tool.
- ☐ Key metrics on a dashboard: traffic, errors, latency, saturation.
- ☐ Distributed tracing connected end to end.
- ☐ Alerts defined, with a named on-call owner.

Resilience

- ☐ Timeouts and retries set on every outgoing call.
- ☐ Circuit breakers or fallbacks for critical dependencies.
- ☐ Load tested at expected peak plus margin.
- ☐ Clear behaviour documented for when this service is down.

Data and security

- ☐ Owns its data; no other service reads its database directly.
- ☐ Backups and restore tested.
- ☐ Secrets managed outside the code.
- ☐ Access limited to the services that need it.

People

- ☐ Runbook written: how to diagnose and fix common issues.
- ☐ API documented and versioned.

TEMPLATE 6

T6

Agent-Ready API Checklist

Run this for every system an AI agent will read from or act on.

Clarity

- ☐ Each action is a small, clearly named endpoint that does one thing.
 - ☐ Inputs and outputs are typed and documented, with examples.
 - ☐ Errors are explicit and explain what to do next.
-

Safety

- ☐ The agent has its own credentials, separate from humans and other services.
 - ☐ Permissions are least-privilege: read and write separated.
 - ☐ Risky actions (payments, refunds, deletes) have limits or require human approval.
 - ☐ Rate limits prevent runaway loops.
 - ☐ Write actions are idempotent: repeating a call does not duplicate the effect.
-

Accountability

- ☐ Every agent action is logged with who, what, when and why.
 - ☐ Actions can be reviewed and reversed.
 - ☐ A kill switch can disable the agent's access instantly.
-

TEMPLATE 7



Architecture **Decision Record** (ADR)

A one-page record of a decision. Six months from now, nobody will remember why you split Billing. This will.

SECTION	CONTENT
<i>ADR number and title</i>	<i>ADR-004: Extract Tracking into its own service</i>
<i>Date and status</i>	<i>Proposed / Accepted / Replaced by ADR-___</i>

Decision makers

Context: what problem are we solving?

Options considered 1. ___ 2. ___ 3. ___

Decision

Why this option

Trade-offs we accept

How we will know it worked

Review date

REFERENCE

Glossary

TERM	PLAIN-ENGLISH MEANING
Monolith	One application containing the whole product, deployed as a single unit.
Modular monolith	A monolith with strict internal boundaries between business modules.
Microservices	A system made of small, independently deployable services, each owning its own data.
Business capability	Something the business does (billing, tracking), used as the unit for splitting.
API	The contract other software uses to talk to a service.
Strangler pattern	Replacing parts of a system gradually by routing traffic to new code piece by piece.
Circuit breaker	A mechanism that stops calling a failing service for a while so failures do not spread.
Eventual consistency	Data across services will match, but not always instantly.
Distributed tracing	Following a single request across every service it touches.
Distributed monolith	Services that cannot be deployed independently. The worst of both worlds.
Idempotent	An action that has the same result whether it runs once or several times.

NEXT STEP

Not sure which side of the fork **you are on?**

DaCodes has spent 10+ years building and scaling software for companies across the Americas, and today builds enterprise AI agents that run in production. We help teams make their systems modular, extract services when it pays off, and get their APIs ready for agents.

Send us your Readiness Scorecard and we will tell you what we would do next.

[Book a call →](#)