Technical Debt

When to Pay It Down vs. When to Just Live With It

Tomas Peluritis

Data Day @ PyCon Lithuania 2026

We all have code in production that makes us cringe.

We all have code in production
that makes us cringe.

And that's... actually okay?

Tomas Peluritis

Head of Data @ Mediatech

AKA Uncle Data

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

https://podcasters.spotify.com/pod/show/duomenu-dede

https://uncledata.substack.com

What Even Is Technical Debt?

Three quick definitions

The textbook answer

Shortcuts taken for speed that cost you later

Three quick definitions

The textbook answer

Shortcuts taken for speed that cost you later

What it actually is

Any gap between your current system and what you wish it was

Three quick definitions

The textbook answer

Shortcuts taken for speed that cost you later

What it actually is

Any gap between your current system and what you wish it was

What matters

Debt that's actively slowing you down or costing you money

I'm no saint...

Code I Wrote That I'm Not Proud Of

(But It's Still Running)

Code I Wrote That I'm Not Proud Of

(But It's Still Running)

ETL Pipeline from 2018

Code I Wrote That I'm Not Proud Of

(But It's Still Running)

ETL Pipeline from 2018

Temporary table that has been dubbed "One Source of Truth"

Code I Wrote That I'm Not Proud Of

(But It's Still Running)

ETL Pipeline from 2018

Temporary table that has been dubbed
"One Source of Truth"

Group of SQL Queries that's over 1k lines long
(and everybody's afraid to touch it)

The Debt Quadrant

Martin Fowler's Technical Debt Framework

The Debt Quadrant

	Reckless	Prudent
Deliberate		
Inadvertent		

The Debt Quadrant

	Reckless	Prudent
Deliberate	**Reckless + Deliberate** *"We don't have time for design"*	
Inadvertent		

The Debt Quadrant

	Reckless	Prudent
Deliberate	**Reckless + Deliberate** *"We don't have time for design"*	
Inadvertent	**Reckless + Inadvertent** *"What's layered architecture?"*	

The Debt Quadrant

	Reckless	Prudent
Deliberate	**Reckless + Deliberate** *"We don't have time for design"*	**Prudent + Deliberate** *"We must ship now and deal with consequences"*
Inadvertent	**Reckless + Inadvertent** *"What's layered architecture?"*	

The Debt Quadrant

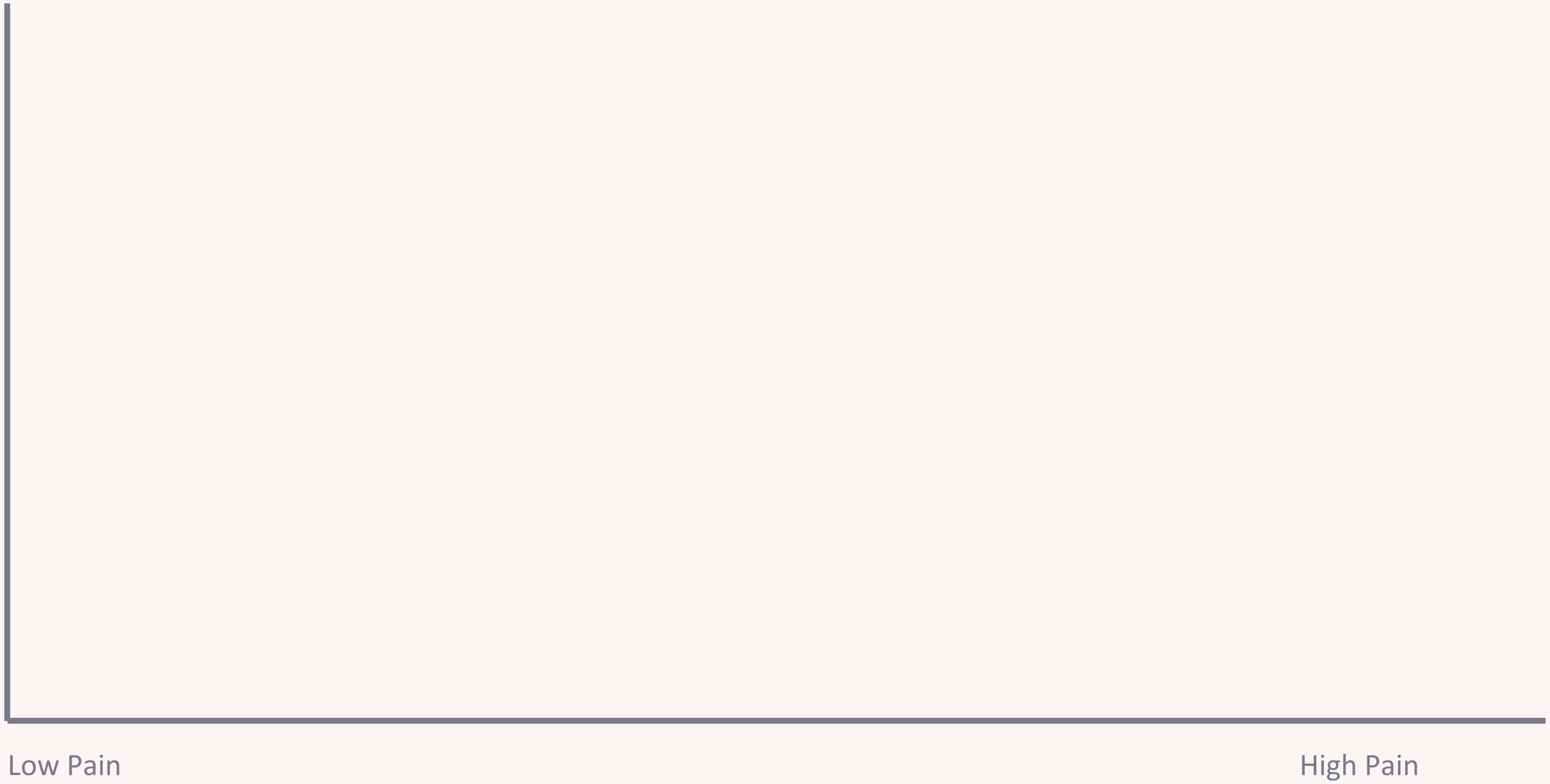
	Reckless	Prudent
Deliberate	**Reckless + Deliberate** *"We don't have time for design"*	**Prudent + Deliberate** *"We must ship now and deal with consequences"*
Inadvertent	**Reckless + Inadvertent** *"What's layered architecture?"*	**Prudent + Inadvertent** *"Now we know how we should have done it"*

The Debt Quadrant

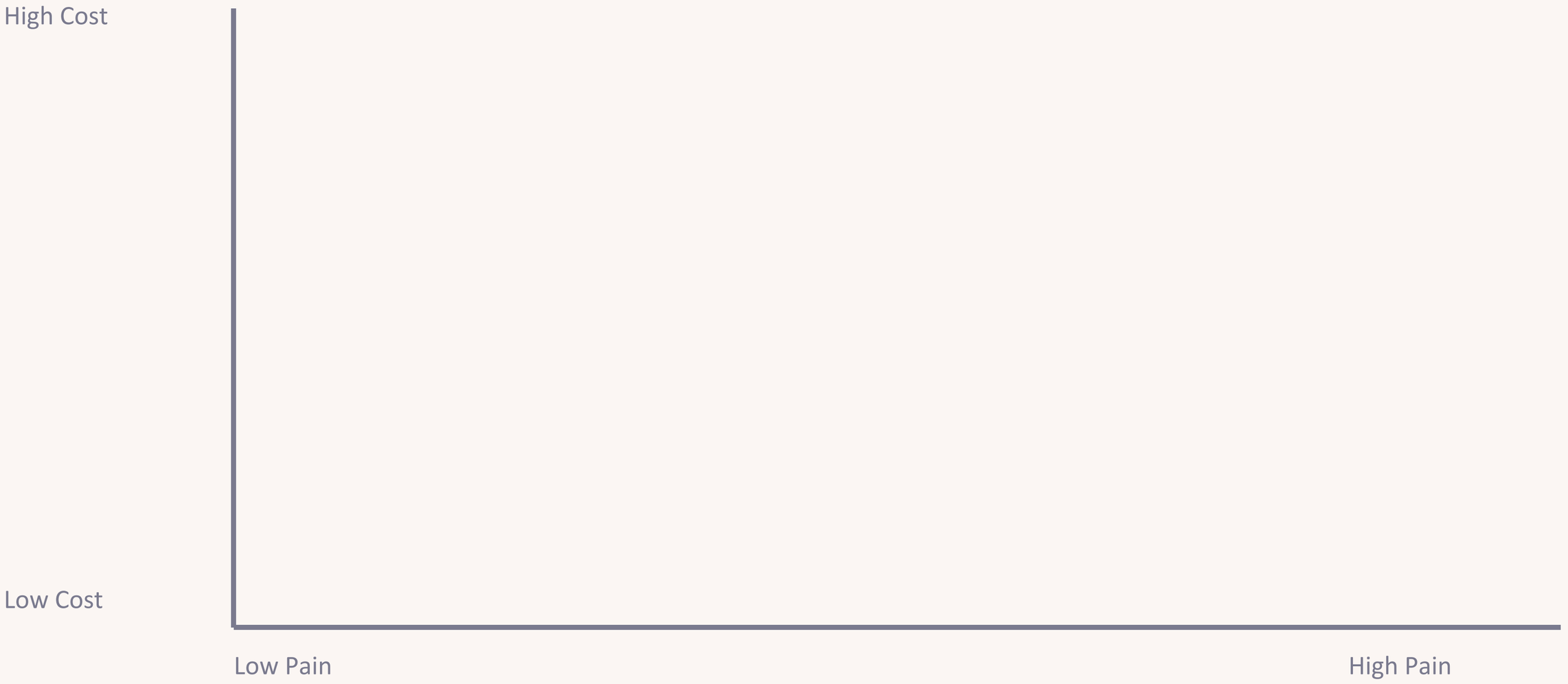
	Reckless	Prudent
Deliberate	**Reckless + Deliberate** *"We don't have time for design"* **Most Teams Live Here**	**Prudent + Deliberate** *"We must ship now and deal with consequences"*
Inadvertent	**Reckless + Inadvertent** *"What's layered architecture?"*	**Prudent + Inadvertent** *"Now we know how we should have done it"*

Not All Debt Is Created Equal

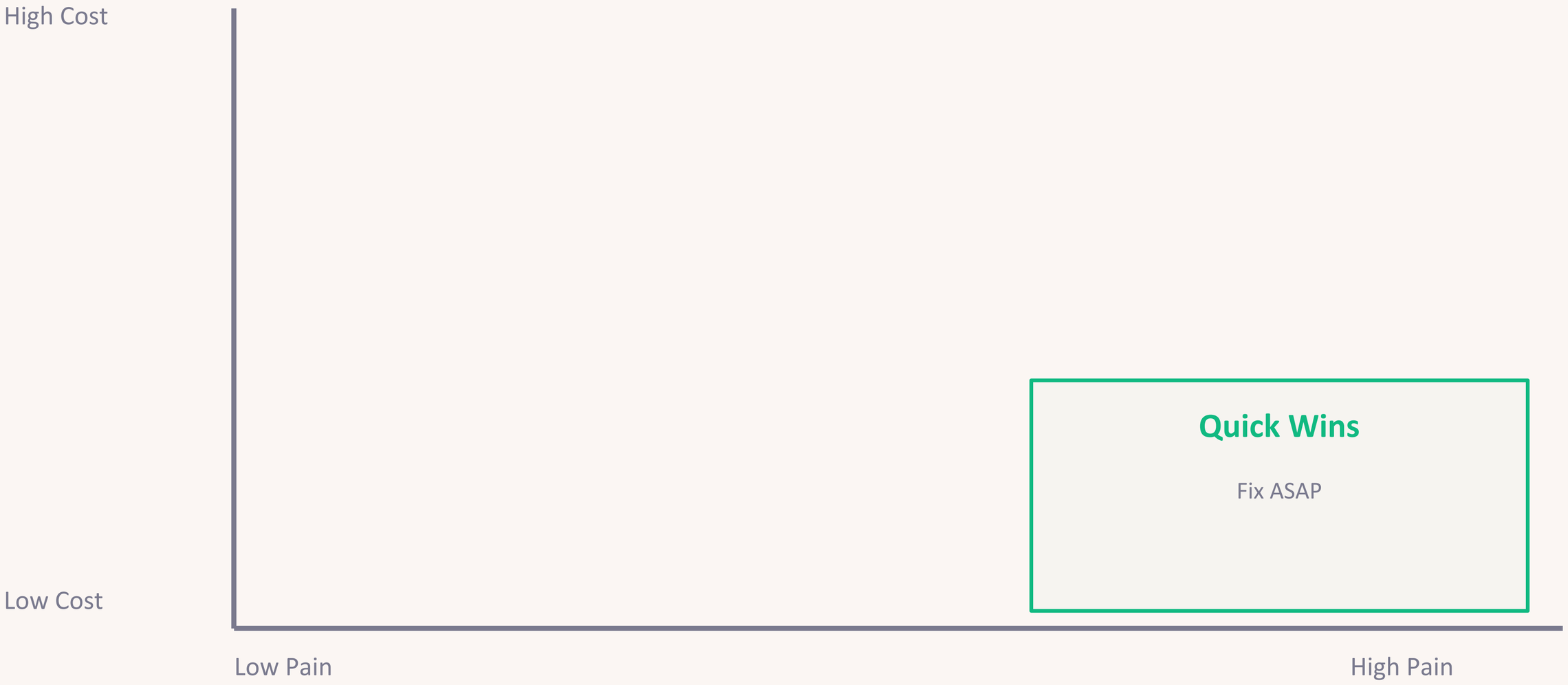
The Pain



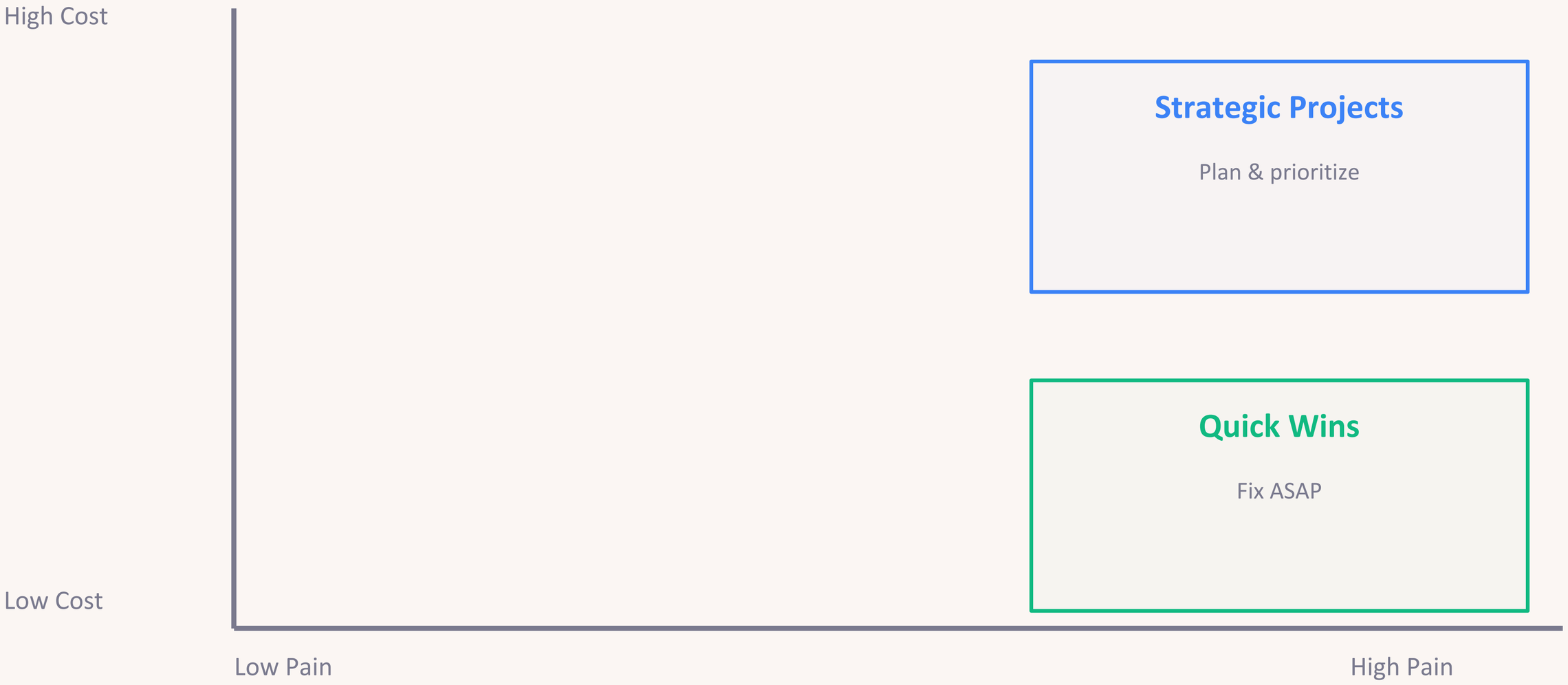
The Pain vs. Cost Matrix



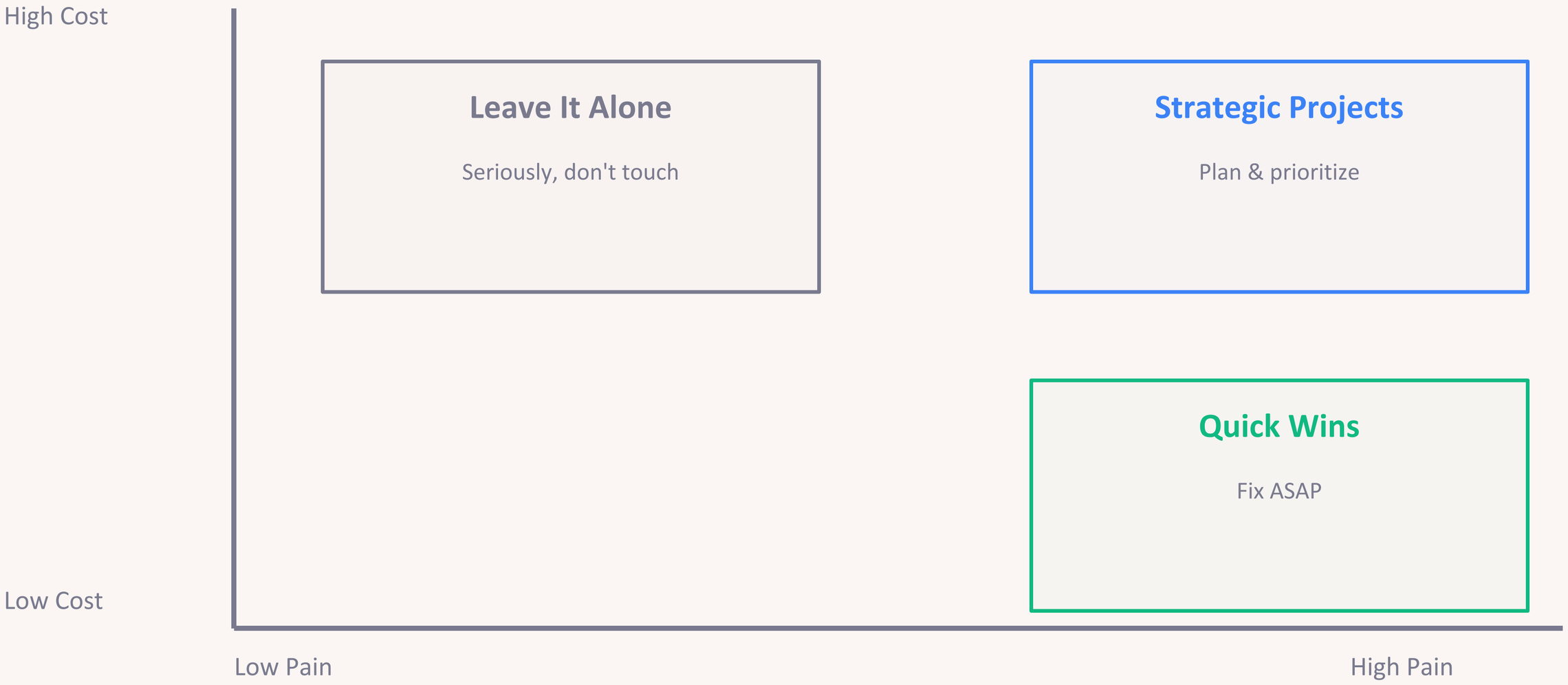
The Pain vs. Cost Matrix



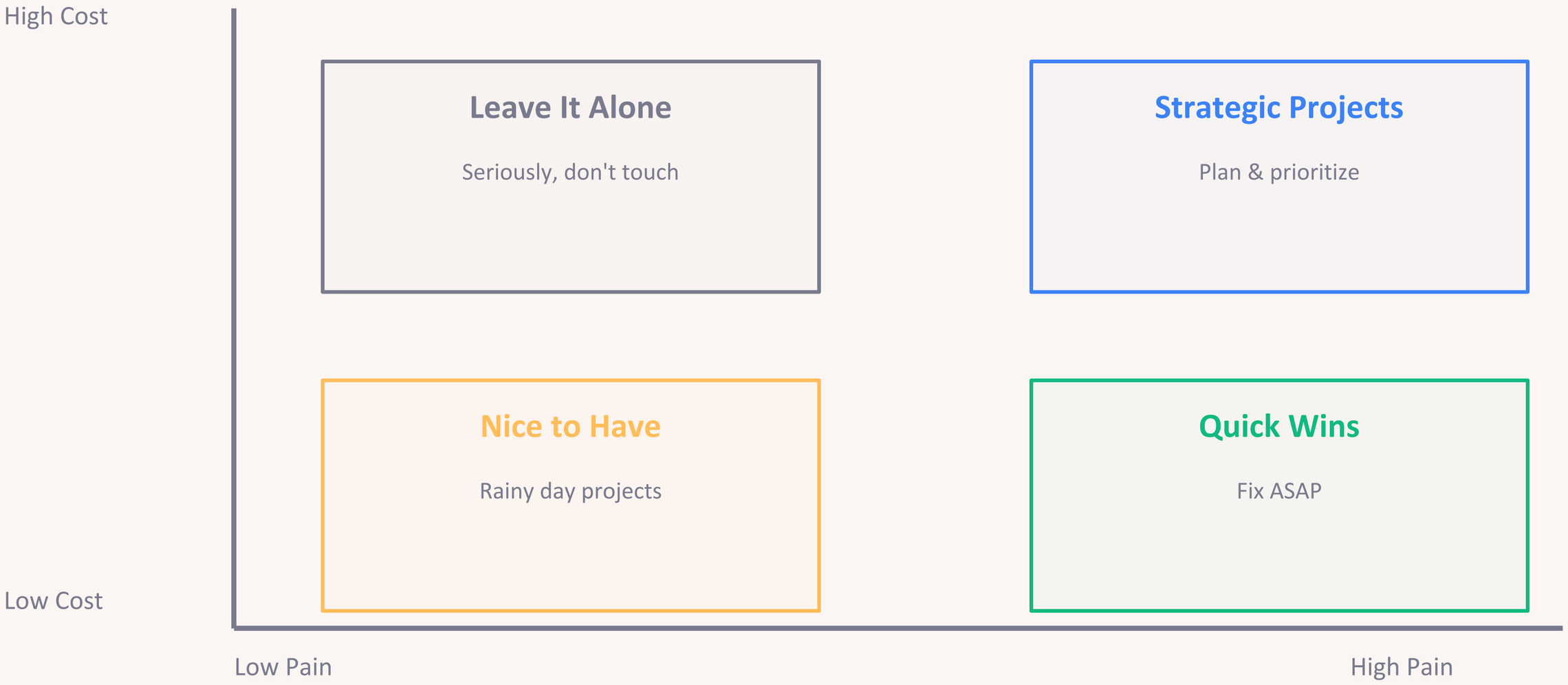
The Pain vs. Cost Matrix



The Pain vs. Cost Matrix



The Pain vs. Cost Matrix



Story Time

Redshift to Snowflake

Redshift → Snowflake

Redshift → Snowflake

Before

- 3 Separate Airflow environments

Redshift → Snowflake

Before

- 3 Separate Airflow environments
- No Dev Env

Redshift → Snowflake

Before

- 3 Separate Airflow environments
- No Dev Env
- Slow Pipelines

Redshift → Snowflake

Before

- 3 Separate Airflow environments
- No Dev Env
- Slow Pipelines

10–12 months to complete

Redshift → Snowflake

Before

- 3 Separate Airflow environments
- No Dev Env
- Slow Pipelines

After

- Dev Env

10–12 months to complete

Redshift → Snowflake

Before

- 3 Separate Airflow environments
- No Dev Env
- Slow Pipelines

After

- Dev Env
- Single Prod Airflow Env

10–12 months to complete

Redshift → Snowflake

Before

- 3 Separate Airflow environments
- No Dev Env
- Slow Pipelines

After

- Dev Env
- Single Prod Airflow Env
- ±20% Cost reduction + easy scaling

10–12 months to complete

Redshift → Snowflake

Before

- 3 Separate Airflow environments
- No Dev Env
- Slow Pipelines

After

- Dev Env
- Single Prod Airflow Env
- ±20% Cost reduction + easy scaling
- Happier engineers

10–12 months to complete

Story Time v2

Legacy Data Pipeline Refactor

Legacy Data Pipeline Refactor

Legacy Data Pipeline Refactor

Before

- Processing done once a week

Legacy Data Pipeline Refactor

Before

- Processing done once a week
- Code was written in PIG

Legacy Data Pipeline Refactor

Before

- Processing done once a week
- Code was written in PIG
- Worked fine without failures for years

Legacy Data Pipeline Refactor

Before

- Processing done once a week
- Code was written in PIG
- Worked fine without failures for years

1-1.5 months to complete

Legacy Data Pipeline Refactor

Before

- Processing done once a week
- Code was written in PIG
- Worked fine without failures for years

After

- Learned PIG

1-1.5 months to complete

Legacy Data Pipeline Refactor

Before

- Processing done once a week
- Code was written in PIG
- Worked fine without failures for years

After

- Learned PIG
- Rewrote in Apache Spark

1-1.5 months to complete

Legacy Data Pipeline Refactor

Before

- Processing done once a week
- Code was written in PIG
- Worked fine without failures for years

After

- Learned PIG
- Rewrote in Apache Spark
- Cleaner code, zero business impact

1-1.5 months to complete

Legacy Data Pipeline Refactor

Before

- Processing done once a week
- Code was written in PIG
- Worked fine without failures for years

After

- Learned PIG
- Rewrote in Apache Spark
- Cleaner code, zero business impact

1-1.5 months to complete

Source got deprecated (as well as my pipeline) in the next 5 months

Legacy Data Pipeline Refactor

Before

- Processing done once a week
- Code was written in PIG
- Worked fine without failures for years

After

- Learned PIG
- Rewrote in Apache Spark
- Cleaner code, zero business impact

1-1.5 months to complete

Source got deprecated (as well as my pipeline) in the next 5 months

Working code that's ugly is often better than "perfect" code that took forever to build.

Story Time v3

Airflow Environment Consolidation

Airflow Environment Consolidation

Airflow Environment Consolidation

Before

- Airflow on EC2 (Docker-in-Docker)

Airflow Environment Consolidation

Before

- Airflow on EC2 (Docker-in-Docker)
- No dev env - test in Jupyter, push to prod

Airflow Environment Consolidation

Before

- Airflow on EC2 (Docker-in-Docker)
- No dev env - test in Jupyter, push to prod
- Residue of docker images pilling in storage, risking crashes

Airflow Environment Consolidation

Before

- Airflow on EC2 (Docker-in-Docker)
- No dev env - test in Jupyter, push to prod
- Residue of docker images pilling in storage, risking crashes
- No IaC - no Terraform, pure ClickOps

Airflow Environment Consolidation

Before

- Airflow on EC2 (Docker-in-Docker)
- No dev env - test in Jupyter, push to prod
- Residue of docker images pilling in storage, risking crashes
- No IaC - no Terraform, pure ClickOps

After (MWAA)

- Single managed environment

Airflow Environment Consolidation

Before

- Airflow on EC2 (Docker-in-Docker)
- No dev env - test in Jupyter, push to prod
- Residue of docker images pilling in storage, risking crashes
- No IaC - no Terraform, pure ClickOps

After (MWAA)

- Single managed environment
- Proper CI/CD pipeline

Airflow Environment Consolidation

Before

- Airflow on EC2 (Docker-in-Docker)
- No dev env - test in Jupyter, push to prod
- Residue of docker images pilling in storage, risking crashes
- No IaC - no Terraform, pure ClickOps

After (MWAA)

- Single managed environment
- Proper CI/CD pipeline
- Auto-scaling (mostly)

Airflow Environment Consolidation

Before

- Airflow on EC2 (Docker-in-Docker)
- No dev env - test in Jupyter, push to prod
- Residue of docker images pilling in storage, risking crashes
- No IaC - no Terraform, pure ClickOps

After (MWAA)

- Single managed environment
- Proper CI/CD pipeline
- Auto-scaling (mostly)
- Better monitoring & alerting

Airflow Environment Consolidation

Before

- Airflow on EC2 (Docker-in-Docker)
- No dev env - test in Jupyter, push to prod
- Residue of docker images pilling in storage, risking crashes
- No IaC - no Terraform, pure ClickOps

After (MWAA)

- Single managed environment
- Proper CI/CD pipeline
- Auto-scaling (mostly)
- Better monitoring & alerting

But we traded old debt for new debt

Airflow Environment Consolidation

Before

- Airflow on EC2 (Docker-in-Docker)
- No dev env - test in Jupyter, push to prod
- Residue of docker images pilling in storage, risking crashes
- No IaC - no Terraform, pure ClickOps

After (MWAA)

- Single managed environment
- Proper CI/CD pipeline
- Auto-scaling (mostly)
- Better monitoring & alerting

But we traded old debt for new debt

- Worker autoscaling bottlenecks during peak hours

Airflow Environment Consolidation

Before

- Airflow on EC2 (Docker-in-Docker)
- No dev env - test in Jupyter, push to prod
- Residue of docker images pilling in storage, risking crashes
- No IaC - no Terraform, pure ClickOps

After (MWAA)

- Single managed environment
- Proper CI/CD pipeline
- Auto-scaling (mostly)
- Better monitoring & alerting

But we traded old debt for new debt

- Worker autoscaling bottlenecks during peak hours
- OOM kills on workers

Migrating away from debt doesn't mean debt-free.

You're trading one set of problems
for (hopefully) better ones.

How to Decide?

A practical decision framework

Five Questions Before You Refactor

Five Questions Before You Refactor

1

Is it actually causing pain?

Not just offending your sensibilities

Five Questions Before You Refactor

1. **Is it actually causing pain?**

   Not just offending your sensibilities

2. **Can you measure the cost?**

   Performance issues, developer time, $$$

Five Questions Before You Refactor

1. **Is it actually causing pain?**

   Not just offending your sensibilities

2. **Can you measure the cost?**

   Performance issues, developer time, $$$

3. **What's the opportunity cost?**

   What else could you build instead?

Five Questions Before You Refactor

1. **Is it actually causing pain?**

   Not just offending your sensibilities

2. **Can you measure the cost?**

   Performance issues, developer time, $$$

3. **What's the opportunity cost?**

   What else could you build instead?

4. **Will it get worse over time?**

   Some debt compounds, some stays stable

Some Debt Gets Worse, Some Stays The Same

Some Debt Gets Worse, Some Stays The Same

Stable Debt

Stays the same over time.

You can live with it.

- One ugly function
- Isolated legacy system
- "Bad" architecture that works

Some Debt Gets Worse, Some Stays The Same

Stable Debt

Stays the same over time.

You can live with it.

- One ugly function
- Isolated legacy system
- "Bad" architecture that works

Compounding Debt

Gets exponentially worse.

Fix it or suffer.

- Lack of tests
- Missing documentation
- Inconsistent patterns spreading

Five Questions Before You Refactor

1. **Is it actually causing pain?**
   Not just offending your sensibilities

2. **Can you measure the cost?**
   Performance issues, developer time, $$$

3. **What's the opportunity cost?**
   What else could you build instead?

4. **Will it get worse over time?**
   Some debt compounds, some stays stable

5. **Can you get stakeholder buy-in?**
   If you can't explain why it matters, maybe it doesn't

How to Actually Get Buy-In for Debt Paydown

Speaking Their Language

What doesn't work

Speaking Their Language

What doesn't work

"The code is messy"

Speaking Their Language

What doesn't work

"The code is messy"

"We need to refactor"

Speaking Their Language

What doesn't work

"The code is messy"

"We need to refactor"

"It's not following best practices"

Speaking Their Language

What doesn't work

"The code is messy"

"We need to refactor"

"It's not following best practices"

"Trust me, it's important"

Speaking Their Language

What doesn't work

"The code is messy"

"We need to refactor"

"It's not following best practices"

"Trust me, it's important"

What does work

Speaking Their Language

What doesn't work

"The code is messy"

"We need to refactor"

"It's not following best practices"

"Trust me, it's important"

What does work

"This costs us $X/month in compute"

Speaking Their Language

What doesn't work

"The code is messy"

"We need to refactor"

"It's not following best practices"

"Trust me, it's important"

What does work

"This costs us $X/month in compute"

"Engineers spend 5 hrs/week on workarounds"

Speaking Their Language

What doesn't work

"The code is messy"

"We need to refactor"

"It's not following best practices"

"Trust me, it's important"

What does work

"This costs us $X/month in compute"

"Engineers spend 5 hrs/week on workarounds"

"It's blocking the feature you want next quarter"

Speaking Their Language

What doesn't work

"The code is messy"

"We need to refactor"

"It's not following best practices"

"Trust me, it's important"

What does work

"This costs us $X/month in compute"

"Engineers spend 5 hrs/week on workarounds"

"It's blocking the feature you want next quarter"

"3 production incidents from this in 2 months"

Common Traps to Avoid

Where teams go wrong

Where Teams Go Wrong

The Perfection Trap

Spending months refactoring working code

"We'll fix everything before adding features"

Where Teams Go Wrong

The Perfection Trap

Spending months refactoring working code

"We'll fix everything before adding features"

The Shiny Object Trap

"Let's rewrite in [Fancy New Tech]!"

Migration for migration's sake

Where Teams Go Wrong

The Perfection Trap

Spending months refactoring working code

"We'll fix everything before adding features"

The Ignore-It Trap

"It's working, don't touch it"

Debt compounds until it explodes

The Shiny Object Trap

"Let's rewrite in [Fancy New Tech]!"

Migration for migration's sake

Where Teams Go Wrong

The Perfection Trap

Spending months refactoring working code

"We'll fix everything before adding features"

The Ignore-It Trap

"It's working, don't touch it"

Debt compounds until it explodes

The Shiny Object Trap

"Let's rewrite in [Fancy New Tech]!"

Migration for migration's sake

The Death March Trap

All debt, all at once

Team burns out, nothing ships

Tech Debt in the Age of AI

How AI changes things

AI Changes the Debt Equation

The Good

Cost to fix goes down

AI Changes the Debt Equation

The Good

Cost to fix goes down

- Understanding legacy code faster

AI Changes the Debt Equation

The Good

Cost to fix goes down

- Understanding legacy code faster
- Generating tests for untested code

AI Changes the Debt Equation

The Good

Cost to fix goes down

- Understanding legacy code faster
- Generating tests for untested code
- Pattern migration at scale

AI Changes the Debt Equation

The Good

Cost to fix goes down

- Understanding legacy code faster
- Generating tests for untested code
- Pattern migration at scale

The Risky

New categories of debt

AI Changes the Debt Equation

The Good

Cost to fix goes down

- Understanding legacy code faster
- Generating tests for untested code
- Pattern migration at scale

The Risky

New categories of debt

- Code nobody fully understands

AI Changes the Debt Equation

The Good

Cost to fix goes down

- Understanding legacy code faster
- Generating tests for untested code
- Pattern migration at scale

The Risky

New categories of debt

- Code nobody fully understands
- Accepted suggestions without review

AI Changes the Debt Equation

The Good

Cost to fix goes down

- Understanding legacy code faster
- Generating tests for untested code
- Pattern migration at scale

The Risky

New categories of debt

- Code nobody fully understands
- Accepted suggestions without review
- 'It works' ≠ 'It's maintainable'

AI Changes the Debt Equation

The Good

Cost to fix goes down

- Understanding legacy code faster
- Generating tests for untested code
- Pattern migration at scale

The Risky

New categories of debt

- Code nobody fully understands
- Accepted suggestions without review
- 'It works' ≠ 'It's maintainable'

The Net Effect

- More things move from "Strategic Projects" to "Quick Wins" on the matrix.
- But AI-generated debt compounds faster because your team didn't write it and doesn't understand it.

Living With Debt Successfully

The part nobody talks about

Managing Debt You Decided to Keep

Document it

If you chose to keep it, write down why and what the risks are

Managing Debt You Decided to Keep

Document it

If you chose to keep it, write down why and what the risks are

Monitor it

Set alerts for when stable debt starts becoming compounding debt

Managing Debt You Decided to Keep

Document it

If you chose to keep it, write down why and what the risks are

Monitor it

Set alerts for when stable debt starts becoming compounding debt

Rotate ownership

Don't let one person be the 'debt expert' who can never leave

Managing Debt You Decided to Keep

Document it

If you chose to keep it, write down why and what the risks are

Monitor it

Set alerts for when stable debt starts becoming compounding debt

Rotate ownership

Don't let one person be the 'debt expert' who can never leave

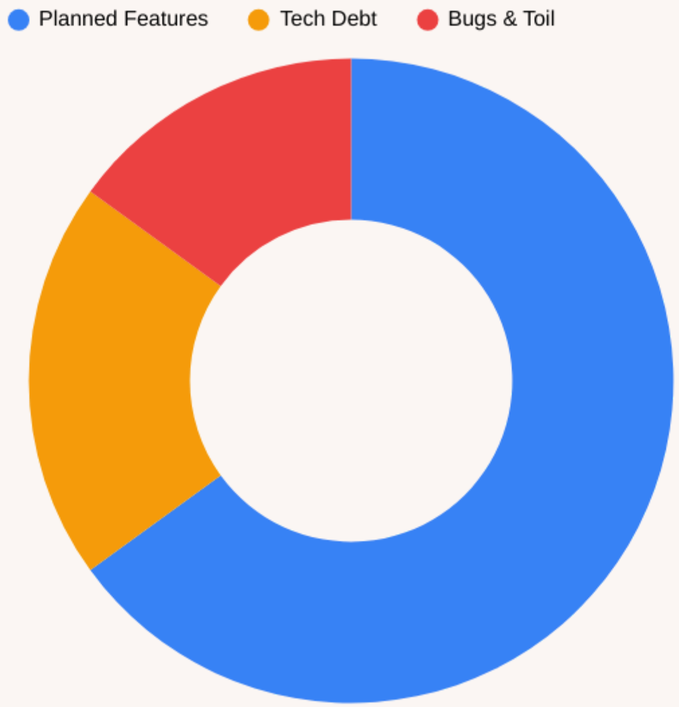
Review quarterly

Pain changes. What was fine 6 months ago might be killing you now

The Debt Budget

15–20%

of each sprint
for debt work



Not a big bang.

A continuous habit.

Small, steady payments beat heroic refactoring every time.

Track the ratio

Tag your work: Planned Feature / Tech Debt / Bug / Operational Toil / Ad-hoc

The split tells you a story about where your team's time actually goes.

The Catalog of Debt

Know what you're living with

Know What You're Living With

What	Pain Level	Cost to Fix	Type	Status
Legacy API	High	2 months	Compounding	Scheduled Q1
Python version upgrade	Low	3 weeks	Stable	Backlog
1k-line SQL query	Medium	1 month	Compounding	Monitoring
No test coverage on ETL	High	Ongoing	Compounding	15% per sprint

Know What You're Living With

What	Pain Level	Cost to Fix	Type	Status
Legacy API	High	2 months	Compounding	Scheduled Q1
Python version upgrade	Low	3 weeks	Stable	Backlog
1k-line SQL query	Medium	1 month	Compounding	Monitoring
No test coverage on ETL	High	Ongoing	Compounding	15% per sprint

You don't need fancy tools. A simple spreadsheet works.

The point is awareness — you can't make good decisions if you don't know what debt you have.

TL;DR

Key Takeaways

1. **Not all debt needs to be paid down**

Key Takeaways

1. **Not all debt needs to be paid down**

2. **Measure pain and cost before refactoring**

Key Takeaways

1. **Not all debt needs to be paid down**
2. **Measure pain and cost before refactoring**
3. **Translate debt into business language**

Key Takeaways

1. Not all debt needs to be paid down
2. Measure pain and cost before refactoring
3. Translate debt into business language
4. Budget for debt work continuously

Key Takeaways

1. Not all debt needs to be paid down
2. Measure pain and cost before refactoring
3. Translate debt into business language
4. Budget for debt work continuously
5. Document and monitor what you keep

Key Takeaways

1. Not all debt needs to be paid down
2. Measure pain and cost before refactoring
3. Translate debt into business language
4. Budget for debt work continuously
5. Document and monitor what you keep
6. AI lowers fixing costs but creates new risks

Technical debt isn't a failure.

It's a tool. Use it wisely.

Q&A

Thank You

Tomas Peluritis

Head of Data @ Mediatech | Uncle Data Newsletter & Podcast