

From Zero to AI

How a small studio rebuilt **the way it ships software**

Andrew “Cheese” Cheeseman

Fractional Chief Architect & Development Lead · cheesesharp.com

45 MINUTES

2 LIVE DEMOS

NO SLIDES ABOUT ROBOTS

What you'll leave with

01

A real architecture

Shared APIs, one admin, throwaway front ends — and why that shape suits events.

02

An automation story

Terraform, pipelines and a build that a single person can run.

03

Two different AIs

Building software with AI is not the same as putting AI in your software.

04

A learning list

What I would learn first if I were starting today.

LIFE OF AN ENGINEER



Thirty years, still in the codebase

I have built the same thing over and over: platforms that outlive the company that ordered them.

IBM

6 YEARS

Microsoft

9 YEARS

ETZ

CTO, £0→£2M

**Benchmark
(Schoders)**

CHIEF ARCHITECT

AWS

SOLUTION ARCHITECT

Graffic Jam

DIRECTOR & LEAD

Hands-on

I am happiest when the terminal is open.
Architecture I cannot build is a drawing.

Small companies

Ten to a hundred engineers. Big enough to
matter, small enough to change.

Brighton

Hove, actually. Twenty-odd years in this scene.

Thirty years, still in the codebase

Reality



Stole snacks
from a tourist

LinkedIn



Proactively sourced
high-value food assets
through direct
stakeholder engagement

Graffic Jam: ten people, very large events

What we were

- A Brighton studio that ran brand for global corporate events.
- Designers, producers, content directors, engineers in the same room.
- Four of us writing software; everyone else needed it yesterday.
- Closed its doors in August 2026. The platform did not.

What that meant technically

- ~~No platform team.~~ ~~No SRE team.~~ ~~No spare capacity.~~
- Deadlines set by an event, not a sprint board.
- The show does not slip. Ever.
- Everything we built had to survive one engineer being ill.

01

What we built

Radius: one platform, many events, and a deliberate split between what we kept and what we threw away.

BESPOKE Frontends

SHARED API / Backends

An event content management system

Registration is where most event tech stops. Everything interesting happens after it: agendas, bookings, media, signage, follow-up.

8

EVENTS IN A YEAR

4

TIME ZONES

6,000+

ATTENDEES

26,000

BOOKINGS HANDLED

300,000+

PHOTOS PUSHED TO SIGNAGE

1.5m

ITEM MEDIA ARCHIVE

Every event is a snowflake. Every back end is not.

What changes every single time

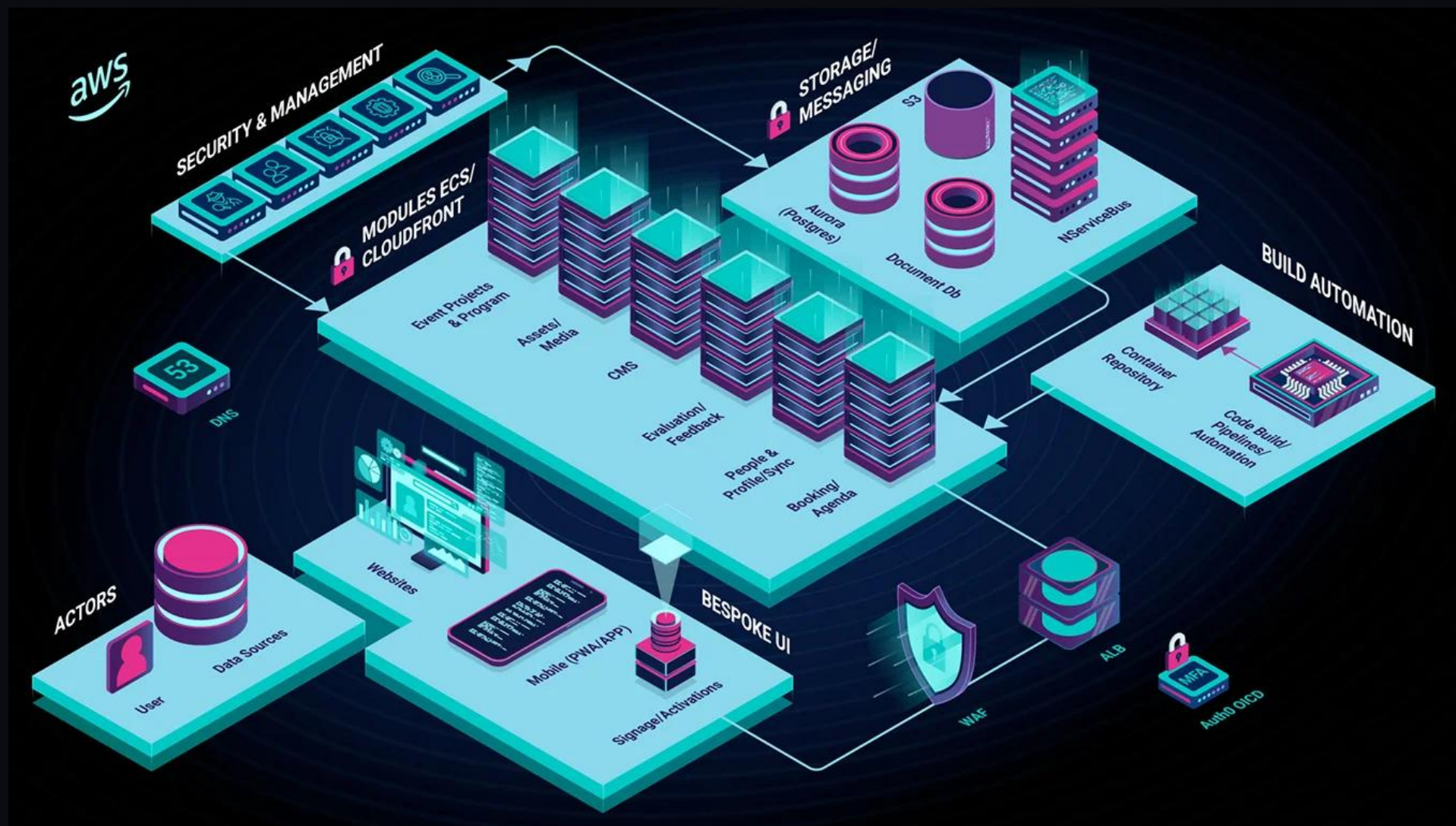
- Brand, tone, layout, language, motion.
- The journey: who sees what, when, and on which screen.
- The client's own opinions, arriving three weeks before the show.

~ 100% `bespoke`

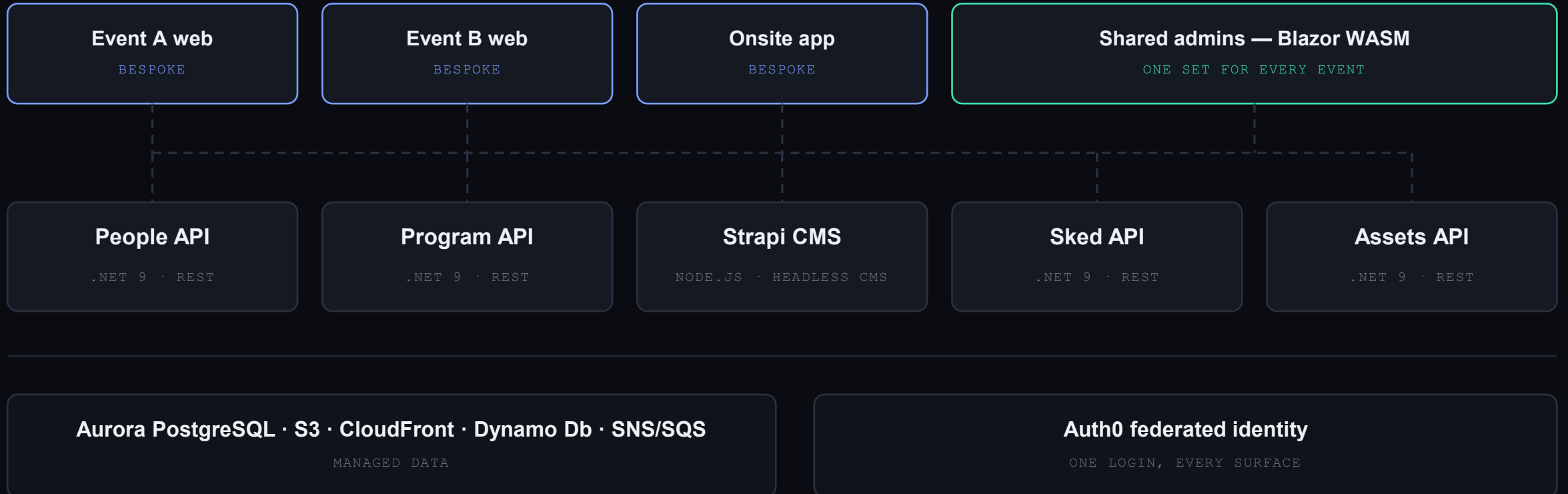
What never changes

- People, organisations, permissions.
- Agendas, sessions, bookings, capacity.
- Assets, media, rendering, distribution.

~ 100% `reusable`



One spine, many faces



contracts published as NuGet packages — the API shape is the contract, and it is versioned

28 repositories · one pipeline · three environments

Build the shared API first, and be strict about it

Domains, not features

- People, Program, Sked, Assets.
- Each one owns its data and its migrations.
- A feature that spans two is a conversation, not a shortcut.

Contracts as packages

- External contract repos, published to a private feed.
- Front ends compile against them.
- Breaking change means a new version, not a broken show.

Boring on purpose

- .NET 9, REST, EF Core, PostgreSQL.
- No cleverness that needs a bus factor of one.
- Every service is the same shape inside.

One set admins to run every event

The staff-facing application is not the product. Build it once, well, and never rebrand it.

Blazor WebAssembly + Fluxor

- C# from database to browser — one language, one team.
- Redux-style state: actions, reducers, effects, per feature.
- A shared Razor component library across all four admins.

Why it paid off

- Producers learn one tool, not one per client.
- A new event is configuration, not a new admin build.
- The unglamorous 80% of the work stopped being rebuilt.

Bespoke front ends, built to be thrown away, **well almost!**

1

React & Next.js against the published contracts

The event site is its own repo, its own life, its own end date.

2

Design-led, not framework-led

The producers and designers set the shape; we make it real.

3

Deleted without ceremony

When the event is over, the front end stops. The data does not.

SHARED MODULE FEDERATION

Owned by the API / Service

People module

Profile and speaker UI, released with the People API.

Agenda / Sked module

Session and schedule UI, released with the Sked API.

Loaded at runtime as remotes; no front end bundles them.

“Disposable” is a compliment here. It is only safe to throw away a front end when the back end is genuinely shared.

The bespoke layer is exactly where AI pays

Stable contract, fresh code

The API shape is fixed and published, so a generated front end has something true to build against. Hallucination has less room to move.

Design to working page, faster

Turning a designer's layout into components was the biggest slice of bespoke time. That slice shrank the most.

Cheap enough to try three

When a first draft is an hour, you stop arguing about which approach to take and just build two.

02

Automate everything

Infrastructure, pipelines and the developer loop — because a team of Four cannot afford a manual step.

Three layers of Terraform, one command each

`builder-foundation`

Accounts, networking, transit gateway, shared VPC. Rarely changes, breaks everything.

`builder-gjd-suite`

The platform: services, data, DNS, certificates. One codebase, three workspaces.

`builder-customer-specific`

Per-client resources. Created for an event, destroyed after it.

Workspaces

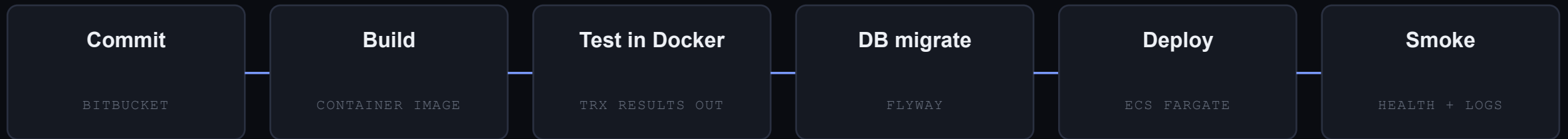
`dev01 · uat01 · prod01`

The workspace name picks the environment file, the DNS prefix, the runtime config.

State in S3, locks in DynamoDB.

The test: a new environment is a workspace and a pipeline run, not a person and a Wednesday.

Commit to running system, without a human in the middle



Pipelines are code too

pipeline-bootstrap, pipeline-setup, pipeline-main: the delivery system is version-controlled like everything else.

Database is part of the deploy

Flyway migrations per domain, run by the pipeline, not by whoever remembered.

Tests run in the image you ship

A Docker target exports the test results, so the thing tested is the thing deployed.

03

Two different AIs

Building software with AI, and putting AI inside software, are different jobs with different risks.

AI as a dev tool code writing and review OR AI as a product feature production

AI that builds the software

- **Runs in:** editor, terminal and CI pipeline.
- **Output:** code, tests, migrations and docs, all diffable and reviewable.
- **Failure mode:** plausible code that compiles but is wrong; caught by review, tests and type checks.
- **Cost and latency:** internal, billed per developer, no user-facing SLO.
- **Reversibility:** remove it and the codebase still builds and ships.

AI inside the product

- **Runs in:** the production request path, on live user data.
- **Output:** an answer a user acts on, with no review before delivery.
- **Failure mode:** confident hallucination served to a customer; needs evals and guardrails.
- **Cost and latency:** per-request token spend and p95 latency count against the SLO.
- **Obligations:** privacy, retention, prompt-injection defence, monitoring and on-call.

we did a lot of the first. be honest about which one you are actually doing

What changes when software gets ~~quick~~ to write

01

More ideas become software

The bar used to be 'is this worth three weeks?'. Now it is 'is this worth an afternoon?'. Far more ideas clear that bar.

02

Each one serves fewer people

Software written for eleven people in one department was never viable. Now it is. Audience size stops being the test.

03

And it is allowed to die

If it took a day to build, nobody defends it for five years. Software becomes a consumable, not an asset.

Bespoke, disposable, and used by eleven people

This is genuinely good news for users and genuinely awkward for how we were taught to build.

What gets easier

Fitting the software to the actual job instead of bending the job to the software.

What gets harder

Ownership. Security review. Knowing what exists. The estate grows faster than the org chart.

What I would insist on

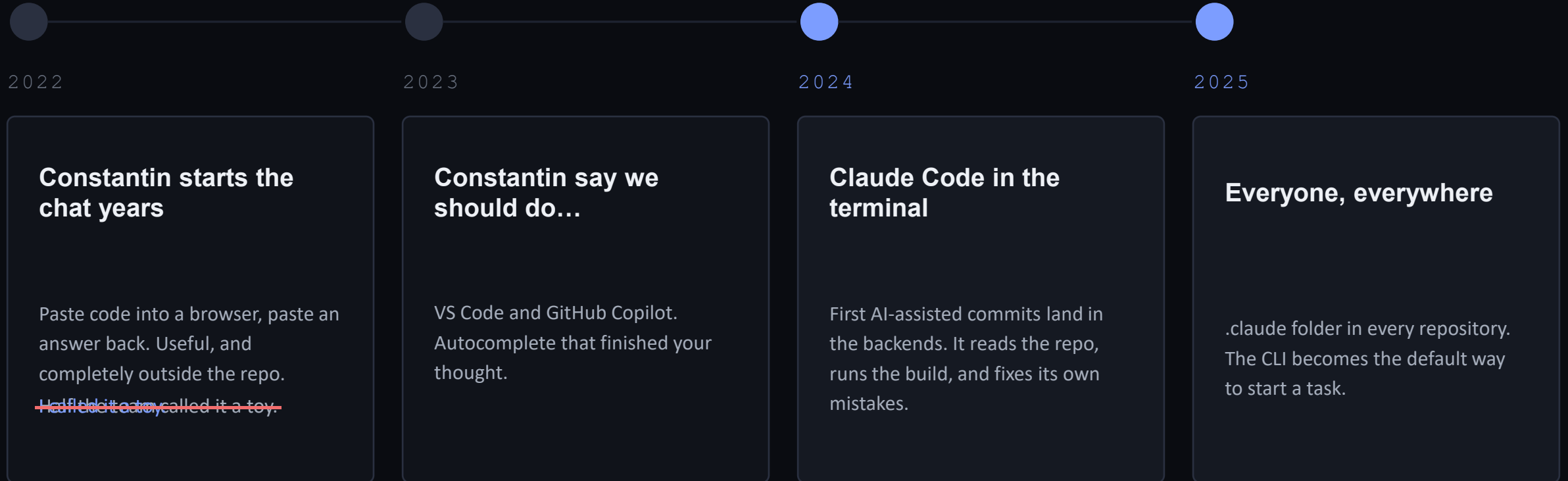
It still goes in source control, still gets a pipeline, still has a name and an owner. Disposable is not the same as untracked.

04

How we got here

Copilot in 2022, a CLI at the end of 2025, and a team that changed its habits in between.

From autocomplete to a colleague in the terminal



Why the terminal beats the app

It can read the whole repository

No more pasting three files and hoping. It goes and looks.

It can run what it wrote

Build, test, read the error, fix it. The loop closes without me.

It works where the work is

Same shell, same git, same branch. No copy-paste seam to lose things in.

It leaves a trail

Commits, diffs, messages — the work is reviewable like anyone else's.

A prompt has four parts; most people skip format

01

Context

Who is writing, for whom?

e.g. Senior .NET developer on the Sked API. Every diff is reviewed by the team.

02

Goal

What exact task needs doing?

e.g. Add a versioned endpoint that returns tomorrow's sessions.

03

Expected

What tone, voice, or rules apply?

e.g. Follow CLAUDE.md. StyleCop clean. xUnit tests for each branch.

04

Format

What's the layout, length, or structure?

e.g. Numbered plan first, then one diff per step. Under 200 lines.

Format is the step 90% of people forget.

Plan first, then build in verified steps

01 PLAN

Agree the change before coding

- **Mode:** plan mode. It reads the repo and proposes; it writes nothing.
- **Input:** the goal, the constraints and the existing code to copy.
- **Output:** numbered steps, each with the test that proves it.
- **Gate:** you edit the plan before approving it.



02 BUILD

Execute one step at a time

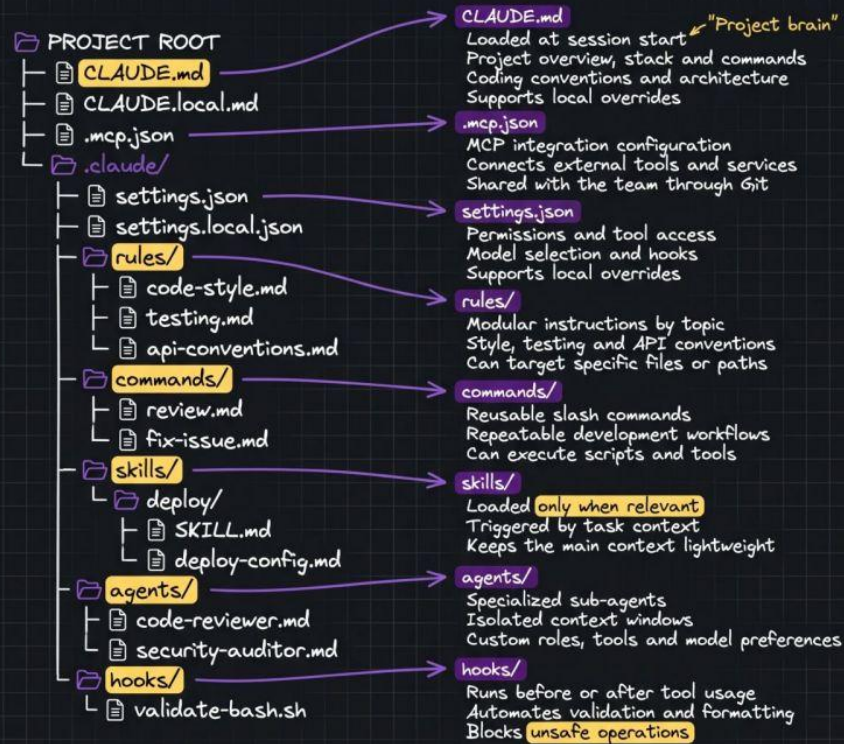
- **Scope:** one plan step per prompt.
- **Feedback:** build and test after every step; paste failures straight back.
- **Checkpoints:** commit each step so any one can be reverted.
- **Reset:** start a fresh session when context drifts; the plan and CLAUDE.md carry over.

changing a plan is a text edit. changing generated code is a review

.claude: the house rules, in the repo

Claude Code

Project Structure



MESSY
PROMPTS

STRUCTURED
PROJECT
CONTEXT

CONSISTENT
CLAUDE CODE
OUTPUT ✓

What it is really doing

- Onboarding a new colleague, in writing.
- Turning tribal knowledge into something enforceable.
- Every repo has one. They get reviewed like code.
- If the AI keeps getting something wrong, the fix is usually this file, not the prompt.

What a developer actually types

```
$ dotnet restore --configfile nuget.config
$ dotnet build -c Release           # warnings are errors
$ dotnet test -c Release --filter Test.Unit
$ docker build --target export-test-results .
$ git tag package -f
$ git tag dev01 -f
$ git push --tags -f
```

```
# same commands locally and in the pipeline
```

House rules in the repo

- Warnings treated as errors.
- StyleCop ruleset, block-scoped namespaces.
- xUnit + FluentAssertions, naming convention drives the filter.
- Local certs scripted — no 'works on my machine' TLS.

Every one of these rules later became a rule the AI had to follow. That is not a coincidence — it is the whole trick.

Skills, reviews, and reviewing yourself

Skills

- Reusable instructions for jobs we do repeatedly: a new API endpoint, a migration, a release note.
- Write the procedure once; stop re-explaining it every session.

Code review

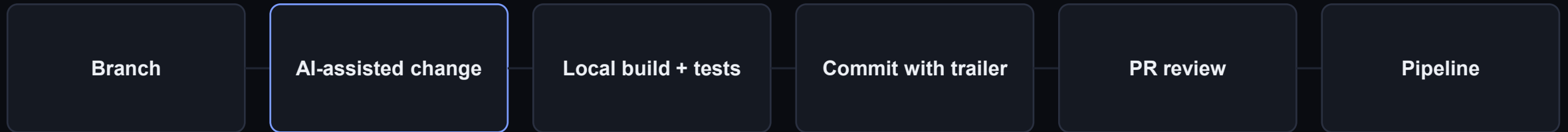
- The model reviews the diff before a human does.
- It is excellent at the boring class: nulls, async holes, missing tests, inconsistent naming.

Self review

- Ask it to review its own work against CLAUDE.md before you even look.
- The second pass catches a surprising amount of the first pass.

Human review never left. It moved: less time on commas, more time on “is this the right thing to build?”

Where AI met the pipeline: git



Git was the interface

No bespoke integration. The AI worked in a branch like anyone else, and the existing pipeline judged it.

Commits say who helped

Co-authored trailers in the message. Two years later you can still see how a change was made.

The build is the referee

Warnings-as-errors and unit tests meant wrong code failed fast, whoever or whatever wrote it.

What improved, and what I will not claim

What genuinely changed

- First drafts appear in minutes, so the work starts at the interesting part.
- The boring 60% — DTOs, mappers, test scaffolding, migrations — mostly stopped being typed.
- Small tools got built. The ones we would previously have muttered about for a year.
- Unfamiliar territory got less scary: a new library became a conversation, not a weekend.
- Fewer 'I'll come back to it' tickets, because coming back became cheap.

What I will not claim

- A percentage. We did not measure it properly, and neither did most people quoting numbers at you.
- That it replaced anyone. Four people did more; four people were still needed.
- That commit counts prove anything. Ours barely moved. The work per commit did.

Why tests and understanding your own code still matter

Plausible beats correct

Generated code is optimised to look right. The compiler catches typos; only tests catch wrong.

You still own it at 3am

When an event is live and something breaks, 'the AI wrote it' is not a diagnosis.

Drift is real

Ask for the same thing twice and get two patterns. Conventions in the repo are what stop the codebase becoming a patchwork.

Review fatigue is the new risk

Volume goes up. If your review habits do not, quality quietly goes down.

Tests were optional-ish when you wrote every line yourself. They are not optional now.

LIVE DEMO 02

What I found useful

- Please AWS, how can we make this cheaper?

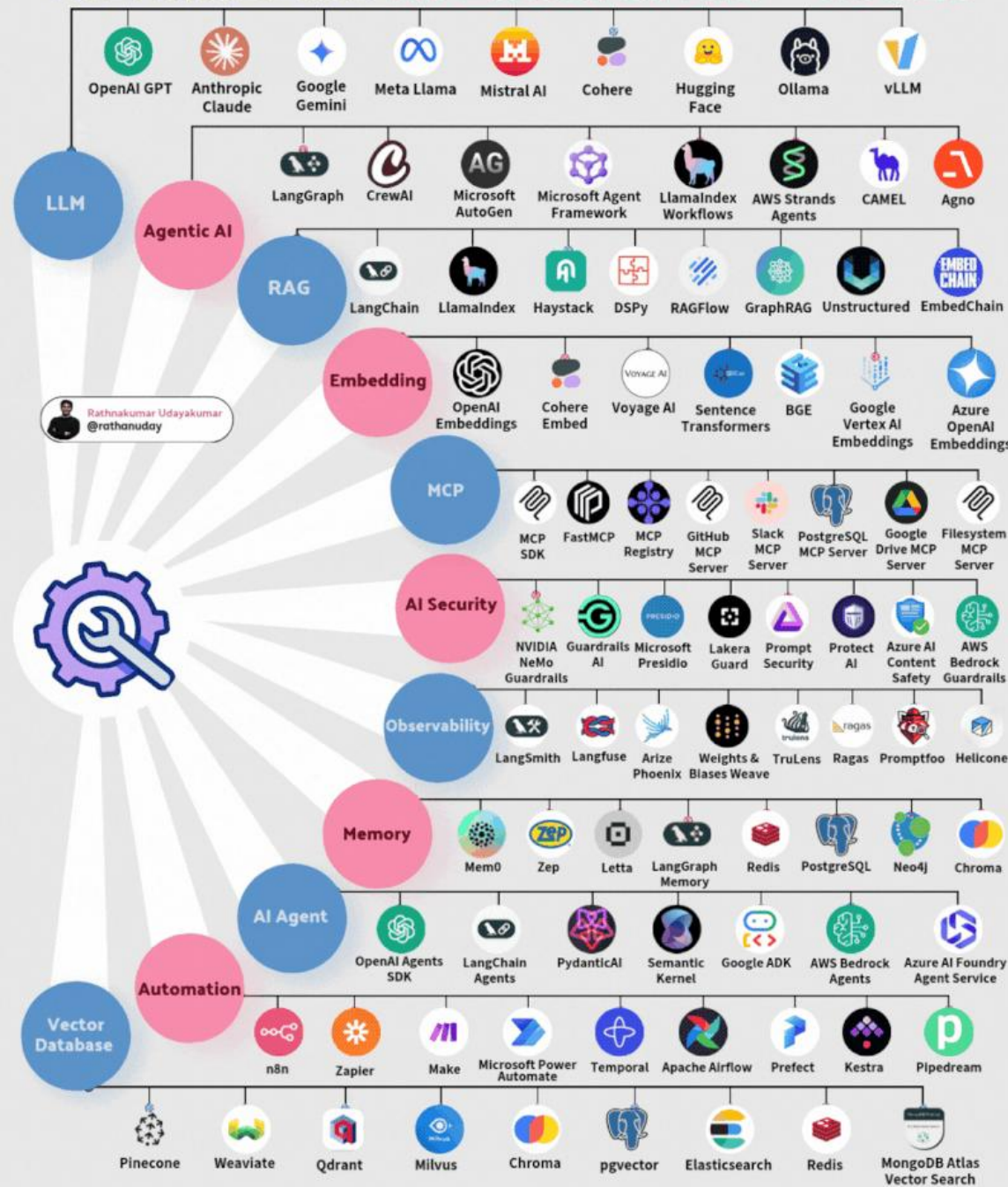
5 minutes – then straight back to the slides

05

If I started today

Four things I would learn first, and one thing I would not bother with.

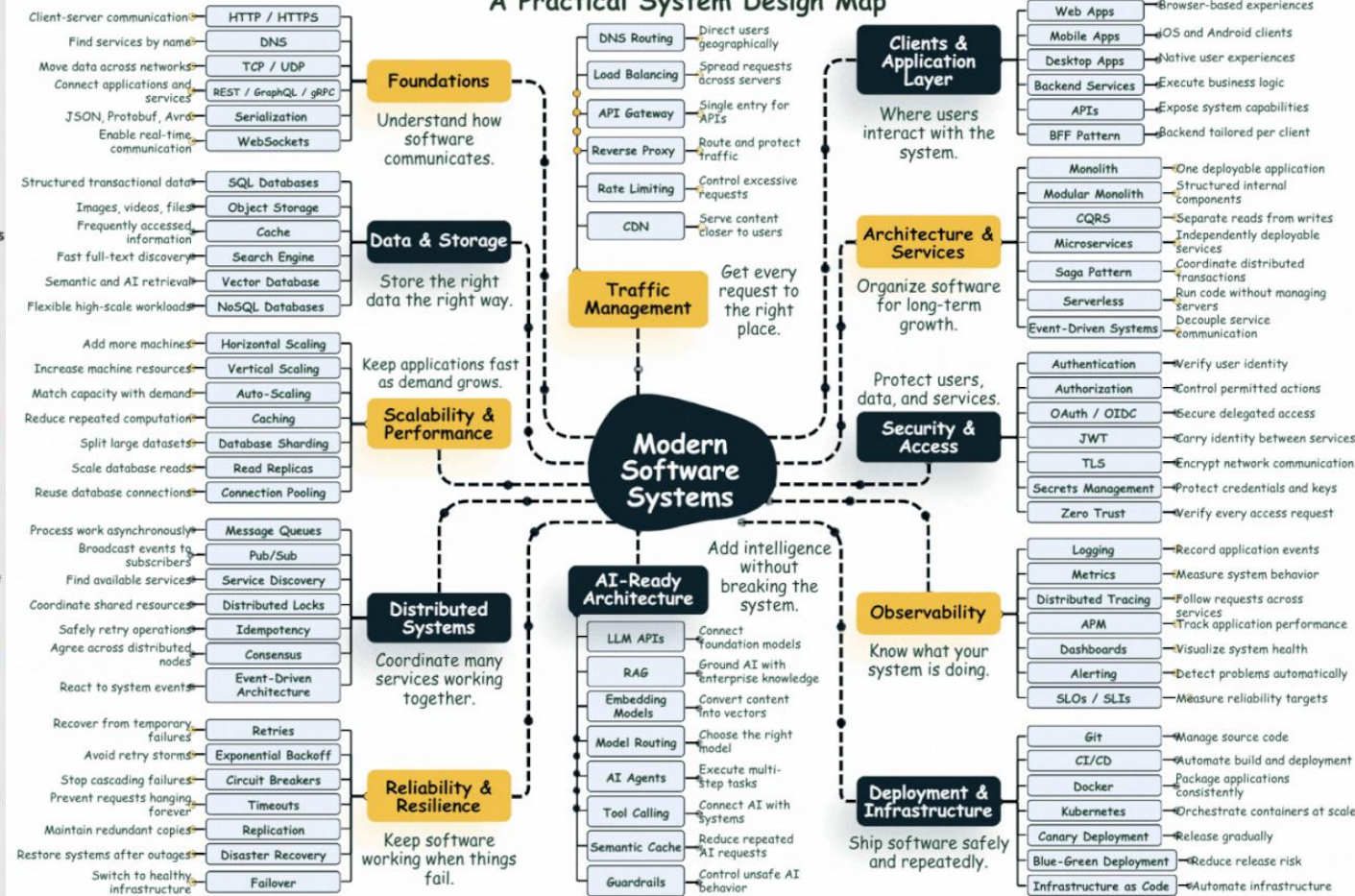
THE MODERN AI ECOSYSTEM - TOOLS



How Modern Apps & Software Are Built

Shalini Goyal

A Practical System Design Map



What I would learn, in this order

01

The fundamentals, once

Google's machine learning crash course. A week of evenings. Enough to stop being sold to, and to know what a model can and cannot do.

02

RAG and pipelines

Retrieval is where most real business value sits: your documents, your data, answerable. Learn chunking, embeddings, evaluation — and why the pipeline matters more than the model.

03

MCP

How tools and data get attached to a model in a standard way. If you only learn one new thing this year, this is the one with the most leverage for an engineer.

04

A2A and connectors

Agents talking to agents, and to the systems you already run. Early, messy, and exactly where integration work is heading.

You do not need to become a data scientist

Unless you are training models, your job is the same job it always was: integration, data, reliability, and knowing what the business actually needs.

What is scarce

Engineers who can wire a model into a real system, safely, with tests and a rollback.

What is not scarce

People who can describe what a transformer is at a dinner party.

Where my 30 years goes

Straight into the bit everyone underestimates: the plumbing, the data, and the failure modes.

Takeaways

1 **Architecture first, then AI**

A shared, versioned contract is what makes generated code safe to throw away.

2 **Put the rules in the repo**

“.claude” directory, hooks, plugins, skills, warnings-as-errors, tests.
The build should be the thing that says no.

3 **Be honest about which AI you mean**

Building with it and shipping it are different jobs, with different risks and different bills.

Five ways organisations have tried to govern AI



01 · Road closed

Prohibition

Staff use ChatGPT on their phones instead



02 · A sign saying "drive carefully"

Paper policy

Every person decides for themselves, and nobody can check



03 · Speed bumps every 50 metres

Gatekeeping

Months-long queue, and the tool has changed by the time it's approved



04 · A single-lane toll booth

Chokepoint

One team becomes everyone's bottleneck and roadmap



05 · Lanes, barriers, cameras

Built-in guardrails

Fast and safe, and you can see what's happening

A.I. TURNS THIS SINGLE
BULLET POINT INTO A
LONG EMAIL I CAN
PRETEND I WROTE.



A.I. MAKES A SINGLE
BULLET POINT OUT OF
THIS LONG EMAIL I CAN
PRETEND I READ.



TOM
FISH
BURNE

I think we may
have a solution to
all our problems.



TOM
FISH
BURNE



— THANK YOU — AND NOW THE INTERESTING PART

Questions

Ask me the awkward ones. Especially about the bits that did not work.

Slides and links will go up after the talk.

Andrew “Cheese” Cheeseman

cheesesharp.com

andrewc@cheesesharp.com

linkedin.com/in/andrew-cheeseman-07a7527

Fractional Chief Architect & Development Lead — two or three days a week.