Garbage ***context,***
Garbage ***code***



Luca Mezzalira

I am confused...





SPEED

Individual




Organization
Holistic Engineering
Vanessa Formicola
People

Context is king



Best practices might not fit your context



Hi!
I’m *Luca Mezzalira*

Software Architect with +20y experience

O’Reilly author

International speaker

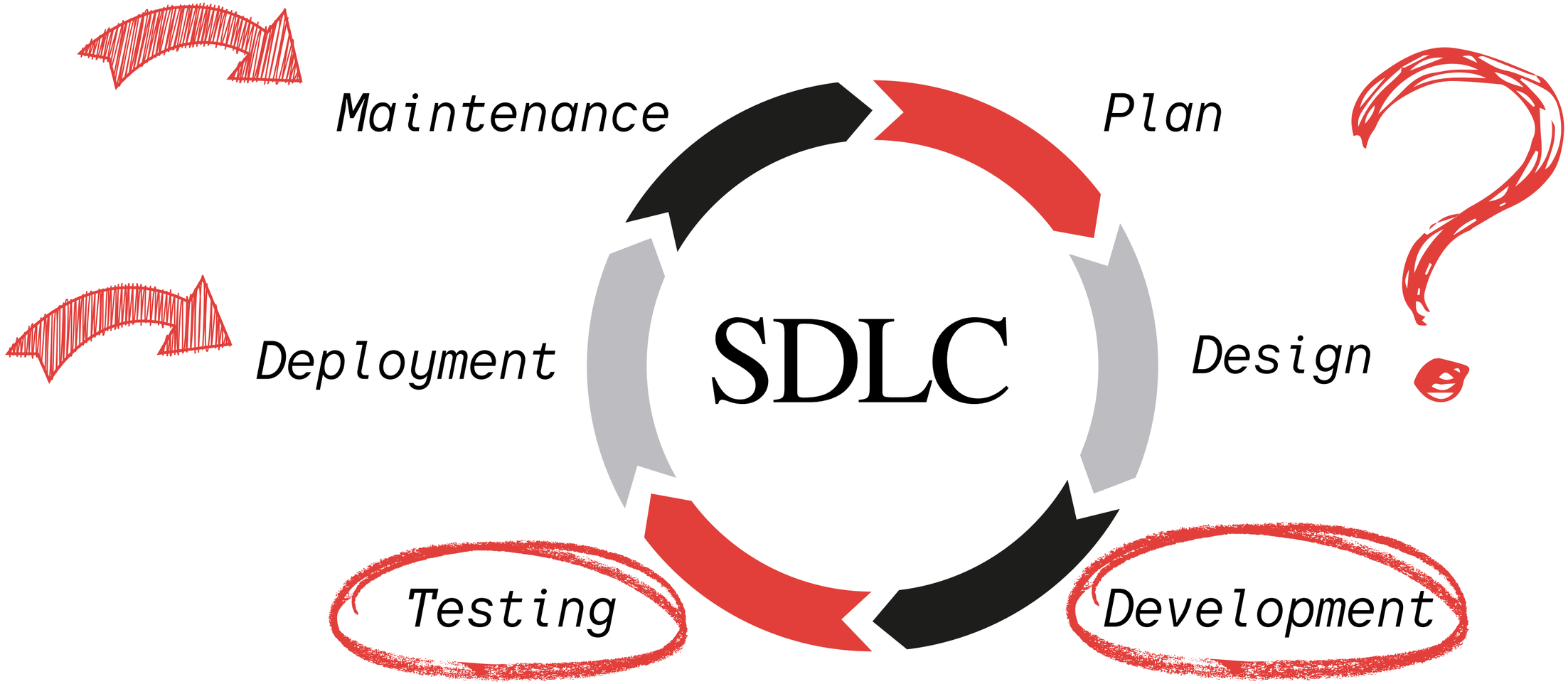
Architecture storyteller



plan

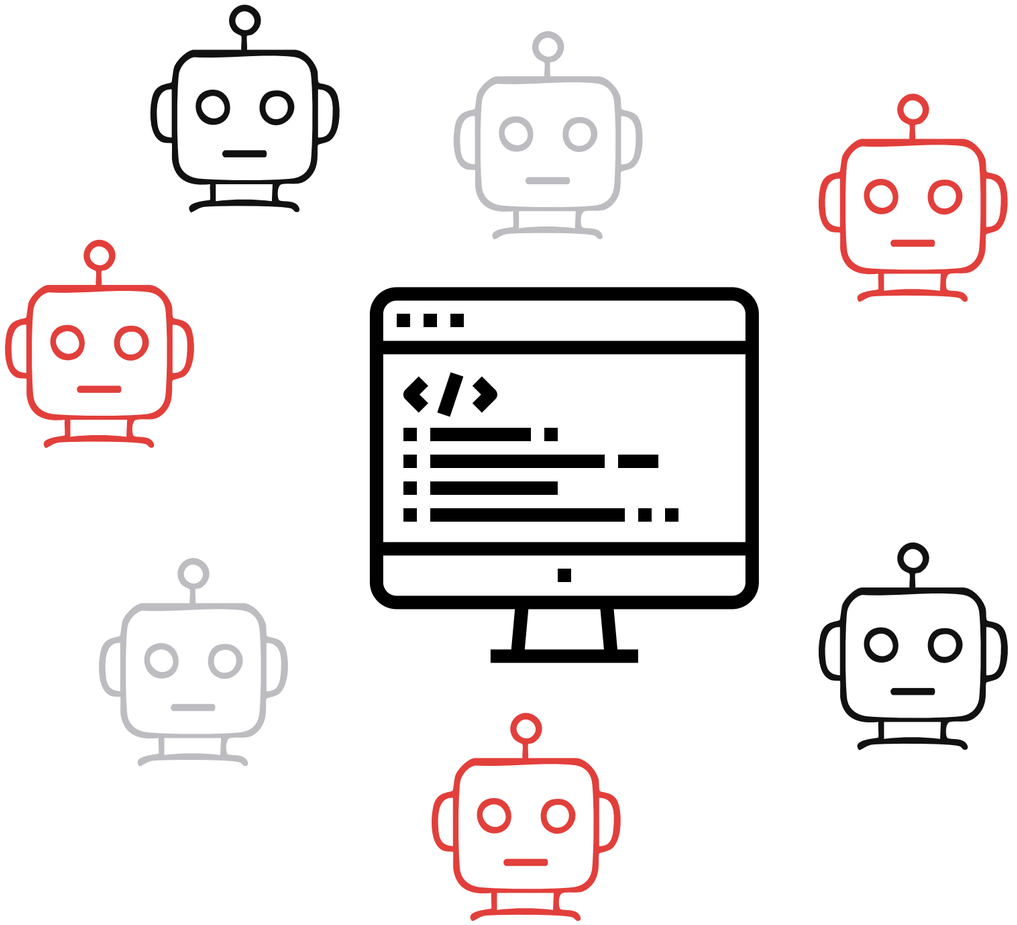
development

review


Maintenance
Plan
Deployment
SDLC
Design
Testing
Development

Spring, Akka, Scala,
.Net framework, AWS, GCP,
Azure, Kafka, Kubernetes,
Serverless, VMs, ...

NOT QUITE



> *Agents are most effective in environments with **strict boundaries** and **predictable structure***

OpenAI

deterministic systems

&

probabilistic systems

Authoring code is different now

Developer-Driven



Prompt Engineering



Context Engineering



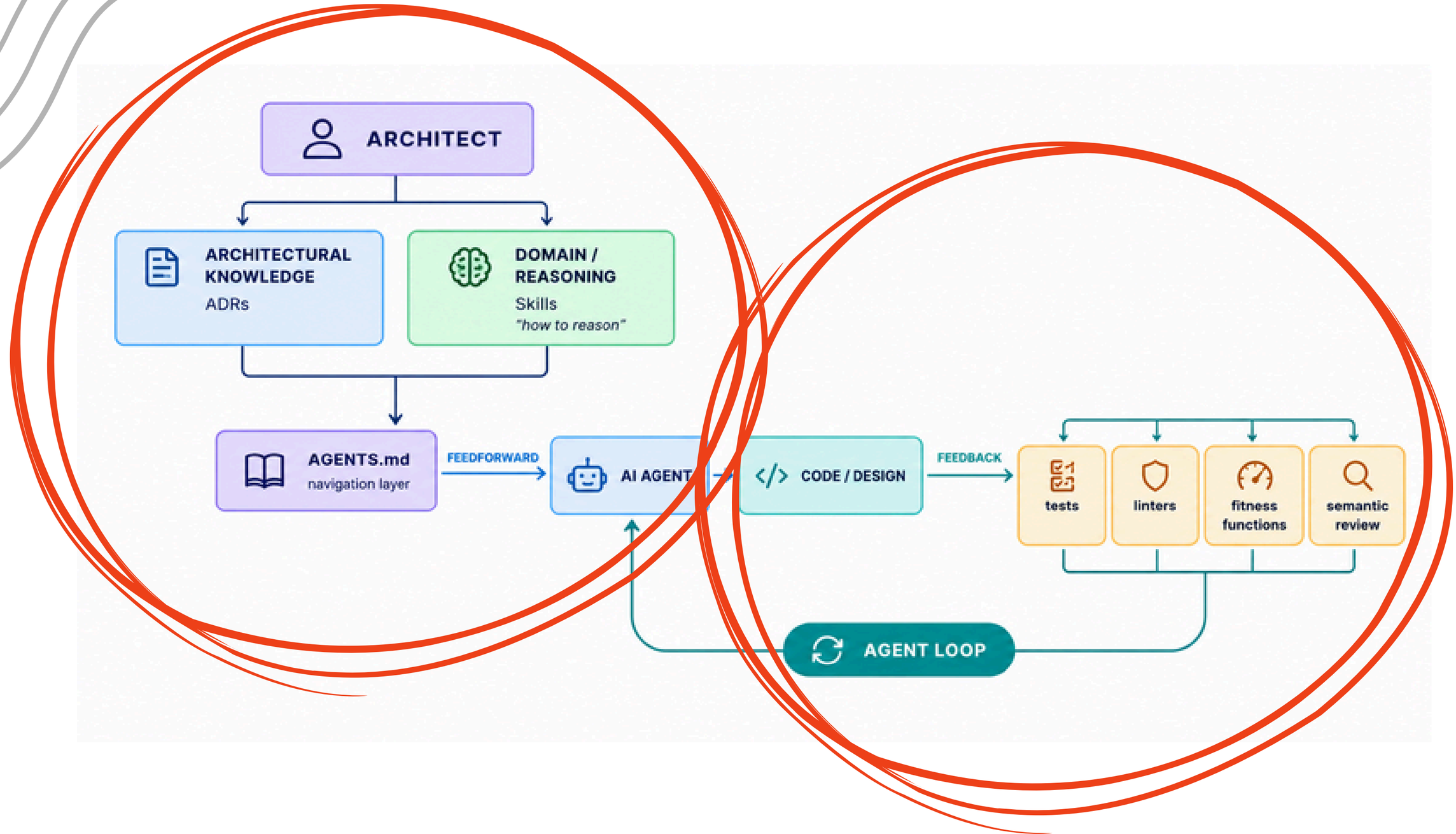
Harness Engineering

Does code still matter 

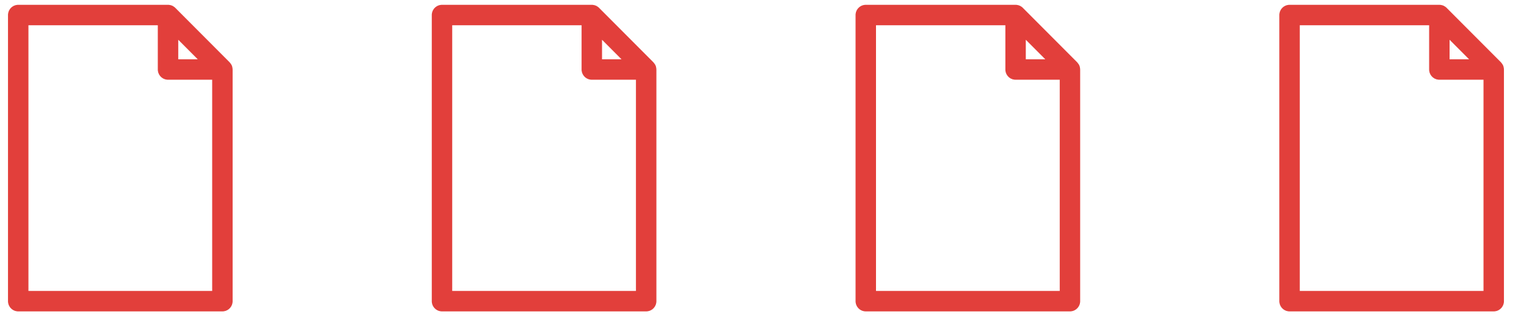
“Does the models pay attention to interface names? ***They they pay attention to the structure****.*

It can allow them to not read the code beneath them, which is both a danger and an advantage. ***As long as the code is consistent, you're okay.”***

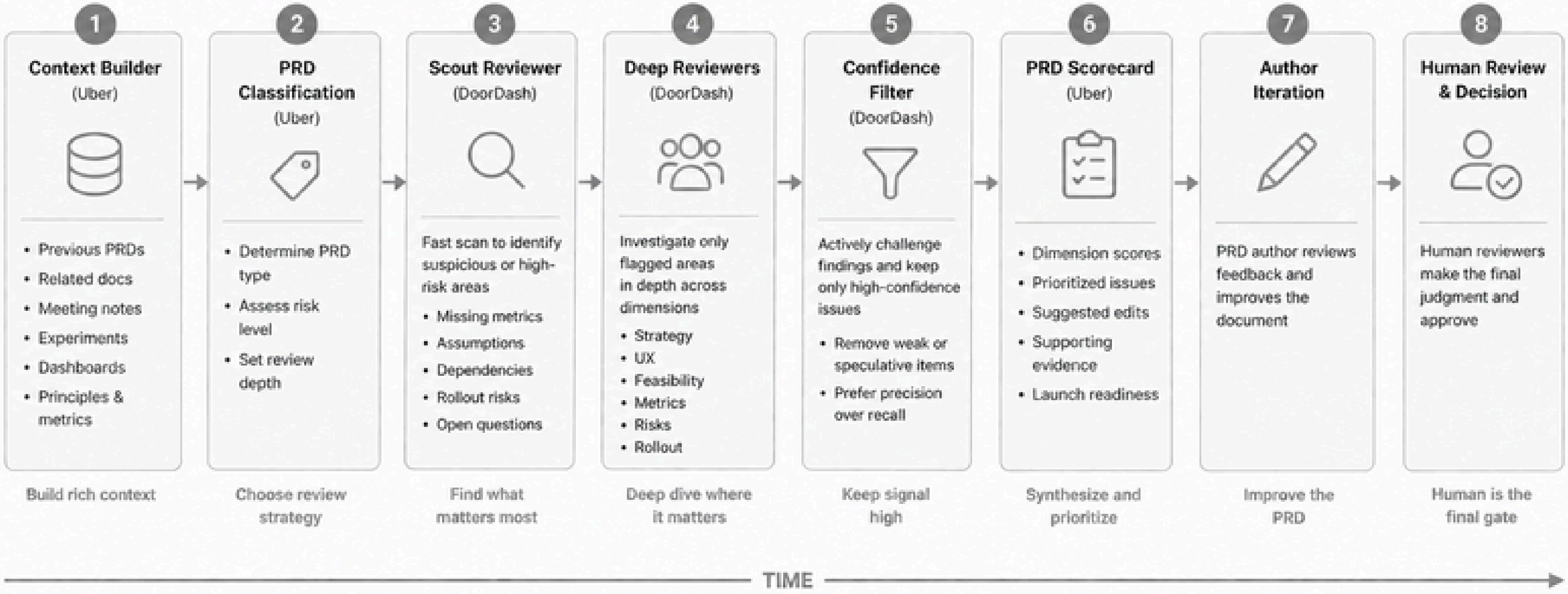
Uncle Bob


ARCHITECT
ARCHITECTURAL KNOWLEDGE
ADRs
DOMAIN / REASONING
Skills
"how to reason"
AGENTS.md
navigation layer
FEEDFORWARD
AI AGENT
CODE / DESIGN
FEEDBACK
tests
linters
fitness functions
semantic review
AGENT LOOP

Product Requirements Document



From the field

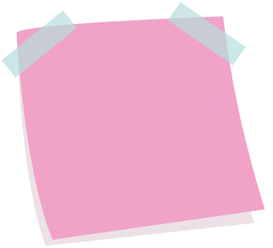


Requirements are an essential part to make sure AI produces ***what you need***

Do you know why







it's ***developer (mis)understanding*** *that's released in production,* ***not the experts' knowledge***



Alberto Brandolini
@ziobrando

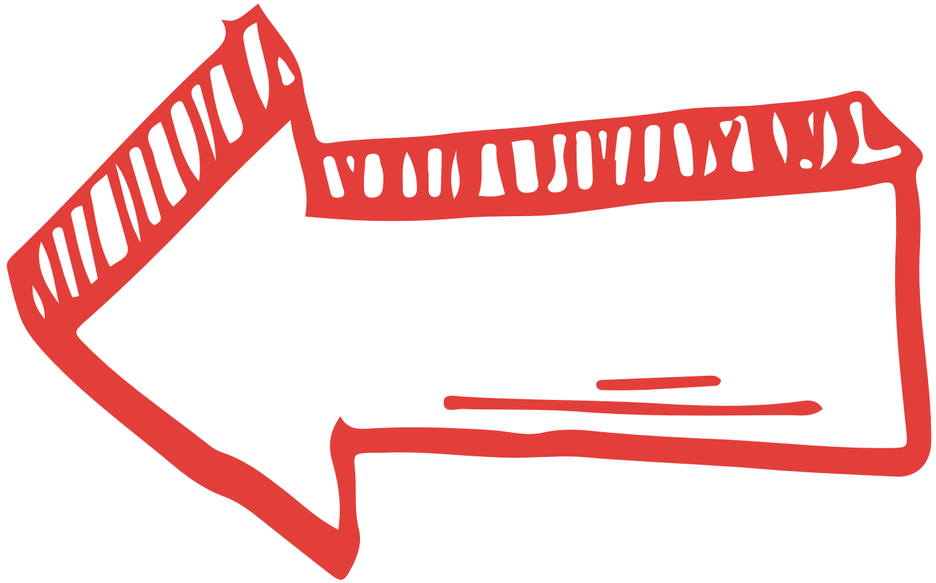
Does architecture still matter




yes

Collaborative methods

- Event Storming
- Event Modeling
- ADRs
- RFCs



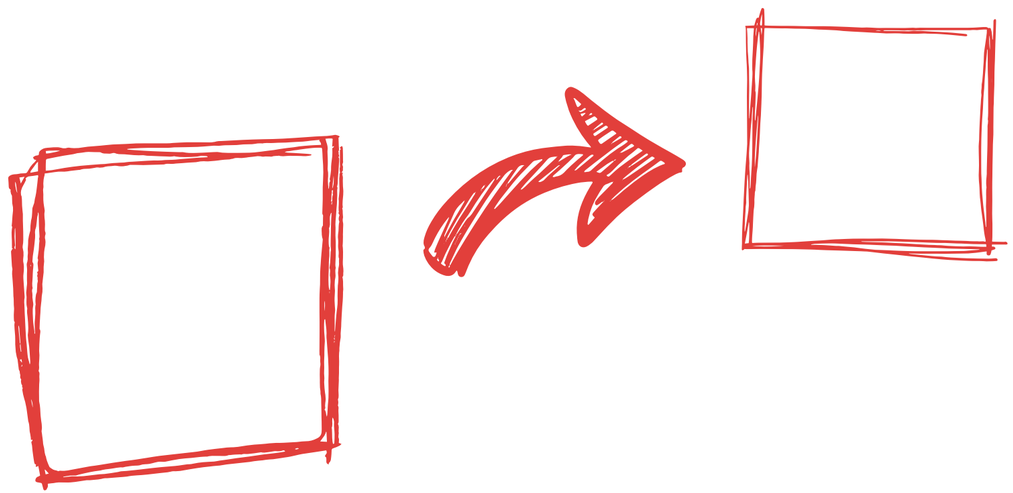
My mental model

- ***Event Storming**: unclear boundaries, large-scale discovery on new domains, system reviews*
- ***Event Modeling**: touching points between teams or at certain lifecycle stages*
- ***ADRs**: records a decision already made and its consequences*
- ***RFCs**: proposal seeking input before a decision*





Human context is NOT in the codebase



Designing a ***socio-technical system***
cannot be reduced to arrows and boxes

*A common repo **nowadays**...*

```
your-repo/
├── AGENTS.md                          # six sections, short, loaded every session
├── HANDOFF.md                         # what the last session left
├── verify.sh                          # one command: lint, types, tests, build
├── rules/                             # one constraint and one reason each
├── skills/
│   ├── adding-endpoints/
│   │   ├── SKILL.md                   # router: instructions and examples
│   │   ├── reference/                 # loaded only when referenced
│   │   ├── scripts/                   # run, never loaded into context
│   │   └── templates/
│   └── adding-consumers/
├── specs/<feature>/                   # spec, plan, acceptance criteria
├── .github/CODEOWNERS                 # paths that always need a human
└── services/payments/AGENTS.md        # nested: nearest file wins
```

AGENTS.md

It tells the agent how to orient itself in the repository, what rules and architectural knowledge apply, and where to find the deeper context it needs before making changes.

AGENTS.md

```
my-project/
├── AGENTS.md
├── .claude/
│   └── skills/
│       ├── docx-contract/
│       │   ├── SKILL.md
│       │   └── scripts/
│       │       └── validate.py
│       └── run-tests/
│           └── SKILL.md
├── services/
│   ├── payments/
│   │   └── AGENTS.md
│   └── auth/
│       └── AGENTS.md
└── src/
```



```
# AGENTS.md

## Project
Payments service. Node.js + TypeScript, Postgres,
deployed on AWS ECS. Monorepo — this service lives
in /services/payments.

## Conventions
- TypeScript strict mode; no `any`
- Tests with Vitest, colocated as *.test.ts
- Conventional Commits for all messages
- Currency is always integer minor units (cents),
  never floats

## What to be careful with
- Never log card numbers, tokens, or PII
- Migrations are forward-only — never edit a shipped one
- Don't touch /services/auth without flagging it first

## Commands
- `pnpm test` — run the suite before any PR
```

```
# {Project Name}

One paragraph on what this repository is, the primary language and framework
with version, and the runtime it targets.

## Setup commands
- Install: `pnpm install`
- Dev: `pnpm dev`
- Test: `pnpm test`
- Test one file: `pnpm test -- path/to/file.test.ts`
- Lint: `pnpm lint`
- Type check: `pnpm typecheck`

## Testing
Unit tests sit next to the code they cover. Integration tests live in
`tests/integration/` and require the local docker-compose stack running.
Never mark a failing test skipped to move on; either fix it or open an ADR
recording why the behaviour changed.

## What done means
A change is done when its tests pass locally, the type checker is clean,
the lint pass is clean, and the diff has been reviewed by the reviewer
role in `harness/`. A change that only compiles is not done.

## Project structure
- `services/` one folder per bounded context, deployed independently
- `packages/` shared libraries, imported by services but never by other packages
- `apps/` micro-frontends, one per user-facing surface
- `infra/` terraform and deploy scripts
- `docs/adr/` architecture decisions, indexed at docs/adr/README.md
- `docs/specs/` per-feature specs, one folder per feature
```

part one

```
# {Project Name}

One paragraph on what this repository is, the primary language and framework
with version, and the runtime it targets.

## Setup commands
- Install: `pnpm install`
- Dev: `pnpm dev`
- Test: `pnpm test`
- Test one file: `pnpm test -- path/to/file.test.ts`
- Lint: `pnpm lint`
- Type check: `pnpm typecheck`

## Testing
Unit tests sit next to the code they cover. Integration tests live in
`tests/integration/` and require the local docker-compose stack running.
Never mark a failing test skipped to move on; either fix it or open an ADR
recording why the behaviour changed.

## What done means
A change is done when its tests pass locally, the type checker is clean,
the lint pass is clean, and the diff has been reviewed by the reviewer
role in `harness/`. A change that only compiles is not done.

## Project structure
- `services/` one folder per bounded context, deployed independently
- `packages/` shared libraries, imported by services but never by other packages
- `apps/` micro-frontends, one per user-facing surface
- `infra/` terraform and deploy scripts
- `docs/adr/` architecture decisions, indexed at docs/adr/README.md
- `docs/specs/` per-feature specs, one folder per feature
```

```
# {Project Name}

One paragraph on what this repository is, the primary language and framework
with version, and the runtime it targets.

## Setup commands
- Install: `pnpm install`
- Dev: `pnpm dev`
- Test: `pnpm test`
- Test one file: `pnpm test -- path/to/file.test.ts`
- Lint: `pnpm lint`
- Type check: `pnpm typecheck`

## Testing
Unit tests sit next to the code they cover. Integration tests live in
`tests/integration/` and require the local docker-compose stack running.
Never mark a failing test skipped to move on; either fix it or open an ADR
recording why the behaviour changed.

## What done means
A change is done when its tests pass locally, the type checker is clean,
the lint pass is clean, and the diff has been reviewed by the reviewer
role in `harness/`. A change that only compiles is not done.

## Project structure
- `services/` one folder per bounded context, deployed independently
- `packages/` shared libraries, imported by services but never by other packages
- `apps/` micro-frontends, one per user-facing surface
- `infra/` terraform and deploy scripts
- `docs/adr/` architecture decisions, indexed at docs/adr/README.md
- `docs/specs/` per-feature specs, one folder per feature
```

```
## Architecture
The standing decisions live in `docs/adr/`. Read `docs/adr/README.md` first
to find the ones that touch your task, then read those in full before making
architectural changes. In short:

- APIs are async by default; sync is used only for read models
- Every service owns its own datastore and never reads another service's tables
- Cross-context data crosses through published events, not shared libraries
- Micro-frontends compose at runtime through module federation; contracts
  live in `packages/mfe-contracts`

For anything beyond these four lines, the ADR is the source of truth and
this file is out of date if they disagree.

## Security
Secrets are read from AWS Parameter Store at boot; never commit `.env` files.
Auth flows for the customer surface live in `packages/auth-web`; the admin
surface uses a separate flow in `packages/auth-admin` and the two never share
tokens. Any change touching authentication requires an ADR, no exceptions.

## Code style
Only the deltas from language defaults. TypeScript strict mode is on;
prefer `unknown` over `any`; no default exports outside `apps/*/pages/`.
Formatting is handled by prettier; do not hand-format.

## Git workflow
Branch from `main`; small commits; conventional commit messages.
Never rebase a branch someone else may have pulled.
Never push to `main` directly.

## Boundaries
- Never modify `packages/*/dist` or anything under `.next/`
- Never touch another service's datastore or migrations
- Never install a new production dependency without an ADR
- Never change files under `docs/adr/` that are in status `accepted`;
  supersede them with a new ADR instead
```

part two

```
## Architecture
The standing decisions live in `docs/adr/`. Read `docs/adr/README.md` first
to find the ones that touch your task, then read those in full before making
architectural changes. In short:

- APIs are async by default; sync is used only for read models
- Every service owns its own datastore and never reads another service's tables
- Cross-context data crosses through published events, not shared libraries
- Micro-frontends compose at runtime through module federation; contracts
  live in `packages/mfe-contracts`

For anything beyond these four lines, the ADR is the source of truth and
this file is out of date if they disagree.

## Security
Secrets are read from AWS Parameter Store at boot; never commit `.env` files.
Auth flows for the customer surface live in `packages/auth-web`; the admin
surface uses a separate flow in `packages/auth-admin` and the two never share
tokens. Any change touching authentication requires an ADR, no exceptions.

## Code style
Only the deltas from language defaults. TypeScript strict mode is on;
prefer `unknown` over `any`; no default exports outside `apps/*/pages/`.
Formatting is handled by prettier; do not hand-format.

## Git workflow
Branch from `main`; small commits; conventional commit messages.
Never rebase a branch someone else may have pulled.
Never push to `main` directly.

## Boundaries
- Never modify `packages/*/dist` or anything under `.next/`
- Never touch another service's datastore or migrations
- Never install a new production dependency without an ADR
- Never change files under `docs/adr/` that are in status `accepted`;
  supersede them with a new ADR instead
```

```
## Architecture
The standing decisions live in `docs/adr/`. Read `docs/adr/README.md` first
to find the ones that touch your task, then read those in full before making
architectural changes. In short:

- APIs are async by default; sync is used only for read models
- Every service owns its own datastore and never reads another service's tables
- Cross-context data crosses through published events, not shared libraries
- Micro-frontends compose at runtime through module federation; contracts
  live in `packages/mfe-contracts`

For anything beyond these four lines, the ADR is the source of truth and
this file is out of date if they disagree.

## Security
Secrets are read from AWS Parameter Store at boot; never commit `.env` files.
Auth flows for the customer surface live in `packages/auth-web`; the admin
surface uses a separate flow in `packages/auth-admin` and the two never share
tokens. Any change touching authentication requires an ADR, no exceptions.

## Code style
Only the deltas from language defaults. TypeScript strict mode is on;
prefer `unknown` over `any`; no default exports outside `apps/*/pages/`.
Formatting is handled by prettier; do not hand-format.

## Git workflow
Branch from `main`; small commits; conventional commit messages.
Never rebase a branch someone else may have pulled.
Never push to `main` directly.

## Boundaries
- Never modify `packages/*/dist` or anything under `.next/`
- Never touch another service's datastore or migrations
- Never install a new production dependency without an ADR
- Never change files under `docs/adr/` that are in status `accepted`;
  supersede them with a new ADR instead
```

```
## Architecture
The standing decisions live in `docs/adr/`. Read `docs/adr/README.md` first
to find the ones that touch your task, then read those in full before making
architectural changes. In short:

- APIs are async by default; sync is used only for read models
- Every service owns its own datastore and never reads another service's tables
- Cross-context data crosses through published events, not shared libraries
- Micro-frontends compose at runtime through module federation; contracts
  live in `packages/mfe-contracts`

For anything beyond these four lines, the ADR is the source of truth and
this file is out of date if they disagree.

## Security
Secrets are read from AWS Parameter Store at boot; never commit `.env` files.
Auth flows for the customer surface live in `packages/auth-web`; the admin
surface uses a separate flow in `packages/auth-admin` and the two never share
tokens. Any change touching authentication requires an ADR, no exceptions.

## Code style
Only the deltas from language defaults. TypeScript strict mode is on;
prefer `unknown` over `any`; no default exports outside `apps/*/pages/`.
Formatting is handled by prettier; do not hand-format.

## Git workflow
Branch from `main`; small commits; conventional commit messages.
Never rebase a branch someone else may have pulled.
Never push to `main` directly.

## Boundaries
- Never modify `packages/*/dist` or anything under `.next/`
- Never touch another service's datastore or migrations
- Never install a new production dependency without an ADR
- Never change files under `docs/adr/` that are in status `accepted`;
  supersede them with a new ADR instead
```

```
## Architecture
The standing decisions live in `docs/adr/`. Read `docs/adr/README.md` first
to find the ones that touch your task, then read those in full before making
architectural changes. In short:

- APIs are async by default; sync is used only for read models
- Every service owns its own datastore and never reads another service's tables
- Cross-context data crosses through published events, not shared libraries
- Micro-frontends compose at runtime through module federation; contracts
  live in `packages/mfe-contracts`

For anything beyond these four lines, the ADR is the source of truth and
this file is out of date if they disagree.

## Security
Secrets are read from AWS Parameter Store at boot; never commit `.env` files.
Auth flows for the customer surface live in `packages/auth-web`; the admin
surface uses a separate flow in `packages/auth-admin` and the two never share
tokens. Any change touching authentication requires an ADR, no exceptions.

## Code style
Only the deltas from language defaults. TypeScript strict mode is on;
prefer `unknown` over `any`; no default exports outside `apps/*/pages/`.
Formatting is handled by prettier; do not hand-format.

## Git workflow
Branch from `main`; small commits; conventional commit messages.
Never rebase a branch someone else may have pulled.
Never push to `main` directly.

## Boundaries
- Never modify `packages/*/dist` or anything under `.next/`
- Never touch another service's datastore or migrations
- Never install a new production dependency without an ADR
- Never change files under `docs/adr/` that are in status `accepted`;
  supersede them with a new ADR instead
```



- *Always do*
- *Ask first*
- *Never do*

Agents.md best practices

1. ***Could I catch the agent breaking it?*** *"Keep the code maintainable" cannot be checked, so it cannot be enforced later either. "Use Decimal, never float, for money" can.*

2. ***Does every don't have a matching do?*** *Forbidding the ORM in route handlers without naming the repository layer leaves the agent choosing blindly between the alternatives you did not mention.*

3. ***Did a real mistake put it there?*** *Agents follow these files literally. A line added speculatively steers every session towards a problem that may never have happened.*

4. ***Is the file still short?*** *Link out to longer material rather than pasting it in.*

MAX between 150-200 lines

Agents.md best practices

1. ***Every rule you add makes every other rule less likely to be followed**, and that has nothing to do with tokens or cost.*

2. ***Write the reason, not just the rule***

3. ***How to structure your rules***

```
base/         applies everywhere in the org
language/     python, typescript
framework/    react, spring
project/      this repo only
```

SKILLS.md

A skill is a directory of instructions and resources that gives the agent a specific capability, packaged so the agent can find it and load it on its own.

SKILLS.md

```
adding-consumers/
├── SKILL.md              # required: frontmatter + instructions
├── references/           # optional: loaded on demand when body links to them
│   ├── idempotency.md
│   └── retries.md
├── scripts/              # optional: executed, never loaded into context
│   ├── scaffold.sh
│   └── check-idempotency.mjs
└── assets/               # optional: templates and lookup data
    └── consumer.ts.tpl
```

```
---
name: adding-consumers
description: Adds a message consumer to a Node service, including idempotency
license: MIT
compatibility: Requires Node 22, pnpm, and Docker for the local broker
metadata:
  author: harness-loop-engineering
  version: "1.0"
---

# Adding Consumers

## When to use this skill

Trigger this skill when a service needs to react to an event published by
another service. Do not use it for synchronous request-response handlers,
which belong in the `adding-endpoints` skill instead.
```

part one

```
---
name: adding-consumers
description: Adds a message consumer to a Node service, including idempotency,
license: MIT
compatibility: Requires Node 22, pnpm, and Docker for the local broker
metadata:
  author: harness-loop-engineering
  version: "1.0"
---

# Adding Consumers

## When to use this skill

Trigger this skill when a service needs to react to an event published by
another service. Do not use it for synchronous request-response handlers,
which belong in the `adding-endpoints` skill instead.
```

```
---
name: adding-consumers
description: Adds a message consumer to a Node service, including idempotency, r
license: MIT
compatibility: Requires Node 22, pnpm, and Docker for the local broker
metadata:
  author: harness-loop-engineering
  version: "1.0"
---
```

```
# Adding Consumers

## When to use this skill

Trigger this skill when a service needs to react to an event published by
another service. Do not use it for synchronous request-response handlers,
which belong in the `adding-endpoints` skill instead.
```

```
## Instructions

Copy this checklist and tick each step as you complete it:

- [ ] Step 1: Scaffold the consumer file and its test
- [ ] Step 2: Implement the handler
- [ ] Step 3: Validate idempotency
- [ ] Step 4: Fix any validation errors and return to step 3
- [ ] Step 5: Continue only once validation passes

**Step 1: Scaffold**

Run: `scripts/scaffold.sh <service-name> <EventName>`

This creates `services/<service-name>/consumers/<EventName>.ts` and the
matching test file, wired into the module registry.
```

part two

```
**Example 2: Consumer that must not double-publish**

Input: The user says "add a consumer for InventoryReserved that emits
a ReservationConfirmed downstream, but never emit twice for the same
reservation."

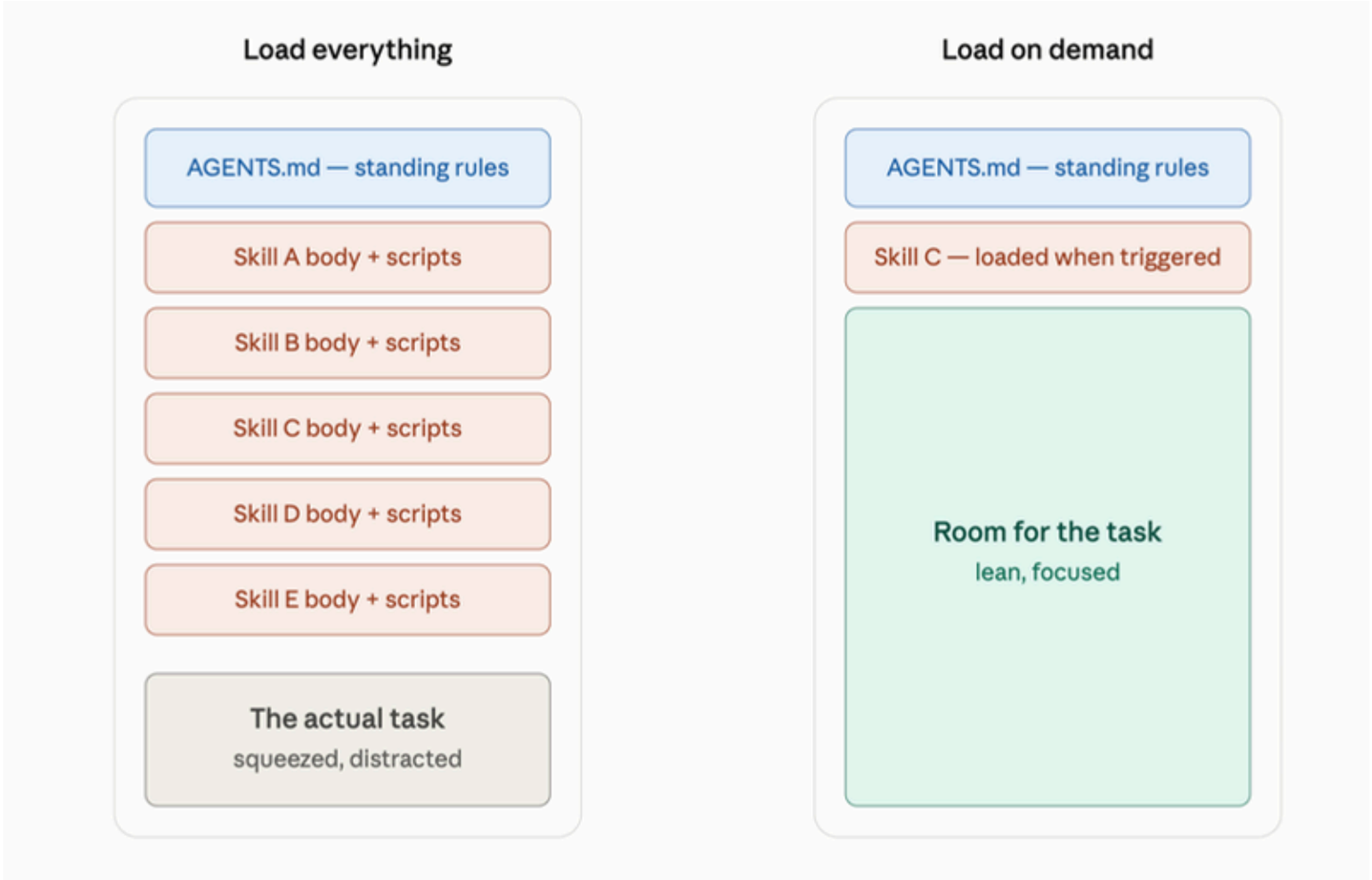
Output: same shape as Example 1, with the downstream `outbox.publish`
call inside the `withIdempotency` wrapper. The tempting wrong version
puts the publish outside the wrapper and races on retry; do not do
this.

## Common edge cases

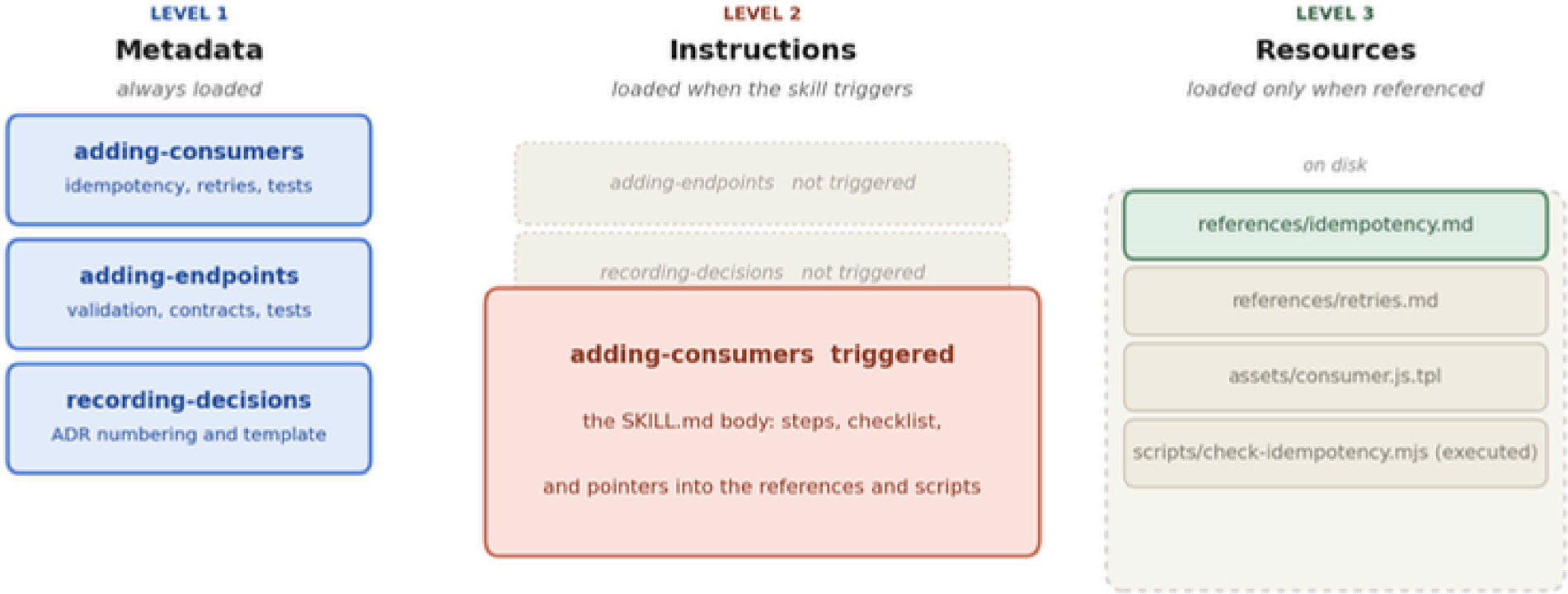
- **Events arriving out of order.** The idempotency store handles
  duplicates but not ordering; if the handler assumes an earlier event
  has been processed, add an explicit precondition check
- **Poison messages.** After three retries the runtime moves the event
  to the dead letter queue; do not add per-consumer retry logic
- **Consumers that call external APIs.** Wrap the call in the
  circuit breaker from `packages/resilience`; direct HTTP calls fail
  the review pass
```

part three

Loading methodology



Skills loading



Rules

- ***A rule is scoped rather than global**: Rules with a glob only fires when the agent touches a file matching that glob. Rules without a glob behave like AGENTS.md and always fire, so scope is the first design decision when writing one*

- ***A rule states a constraint, not a procedure**: Rules describe what must always be true regardless of what the agent is doing.*

- ***A rule carries its reason**. The reason is what turns the rule from a rigid command into something an agent can apply intelligently, and it is what lets you delete the rule the day the reason stops being true*

```
---
description: Every consumer must be idempotent because retry is automatic
globs: services/*/consumers/**/*.ts
alwaysApply: false
---

# Consumers are idempotent

When a message consumer handles an event, it shall complete the side
effect at most once per messageId, because the broker retries on any
error and duplicate side effects corrupt downstream state.

## Signals of violation

- Any consumer that mutates state without checking the idempotency store
- Any consumer that publishes to the outbox outside the idempotency
  wrapper
- Any test that asserts a side effect happened twice as if that were
  correct behaviour

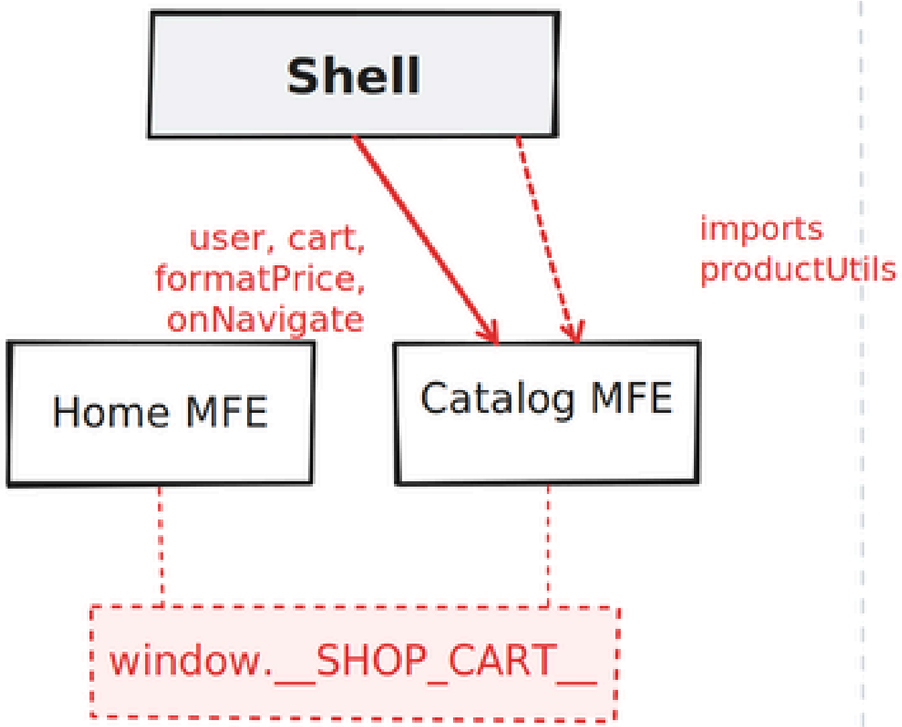
## How to satisfy it

Wrap the handler in `withIdempotency` from `packages/idempotency`.
The wrapper takes the messageId from the event envelope and returns
early if it has been seen.
```

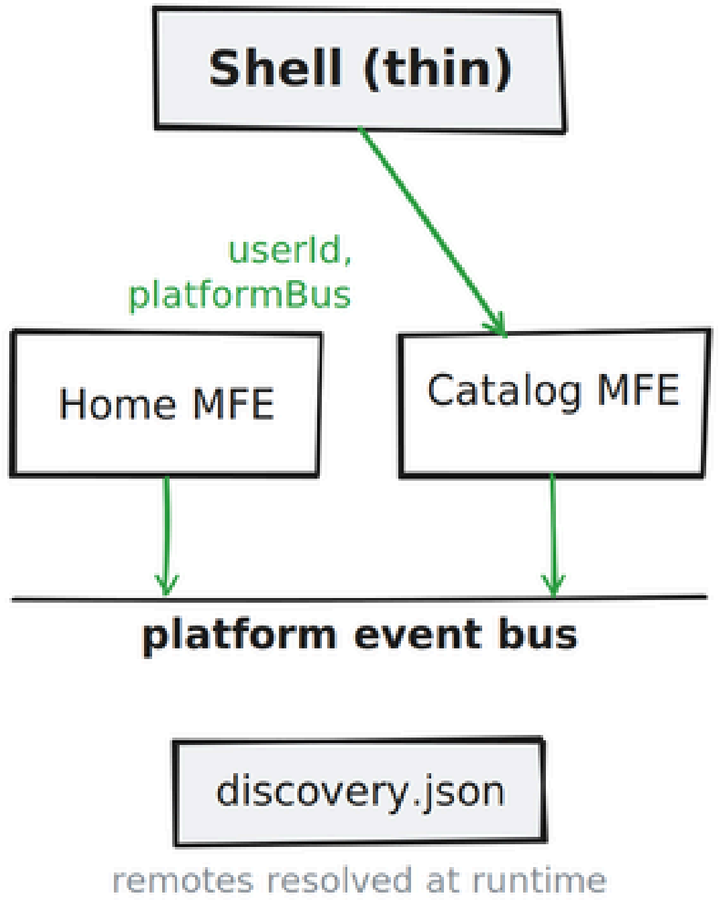
```
---
description: Never commit secrets or credentials to the repository
---

Any file that would introduce an AWS key, a database password, a
token or any other credential shall not be committed, because
secrets in the git history are permanent even after removal and
must be rotated at the source rather than expunged from the log.
Use AWS Parameter Store for runtime secrets and .env.example for
development placeholders.
```

It always applies



tangled at the seams

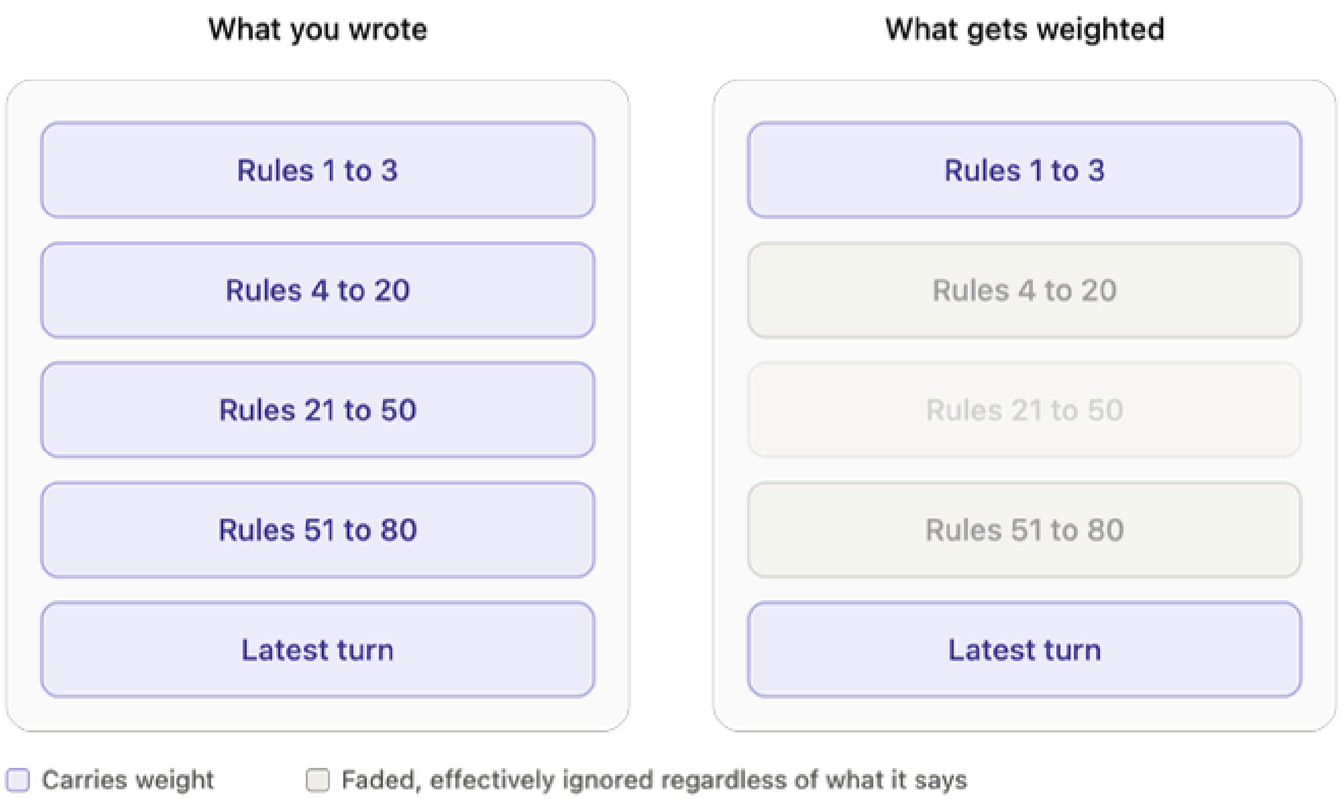


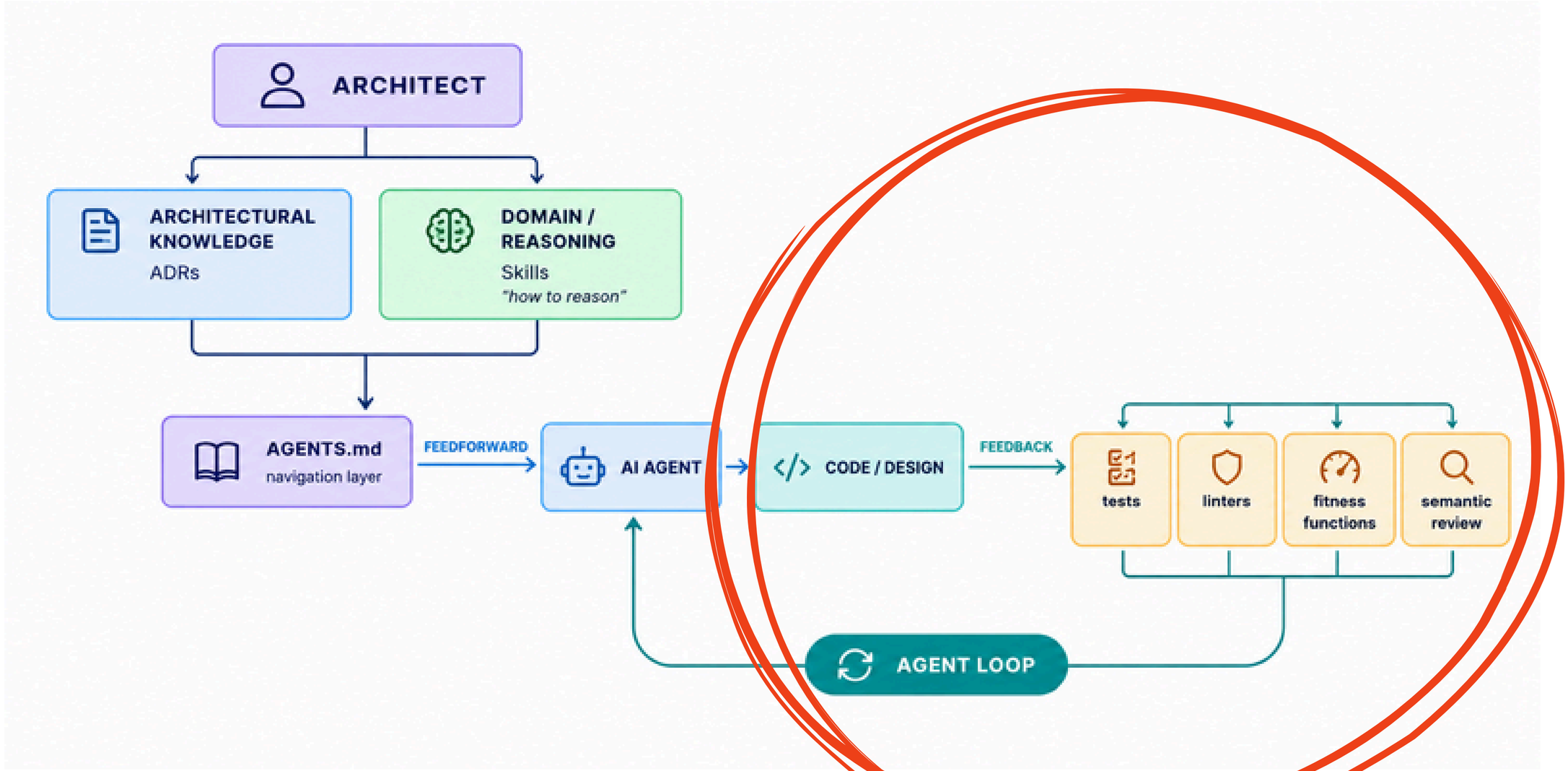
clean, independent islands

The context ceiling

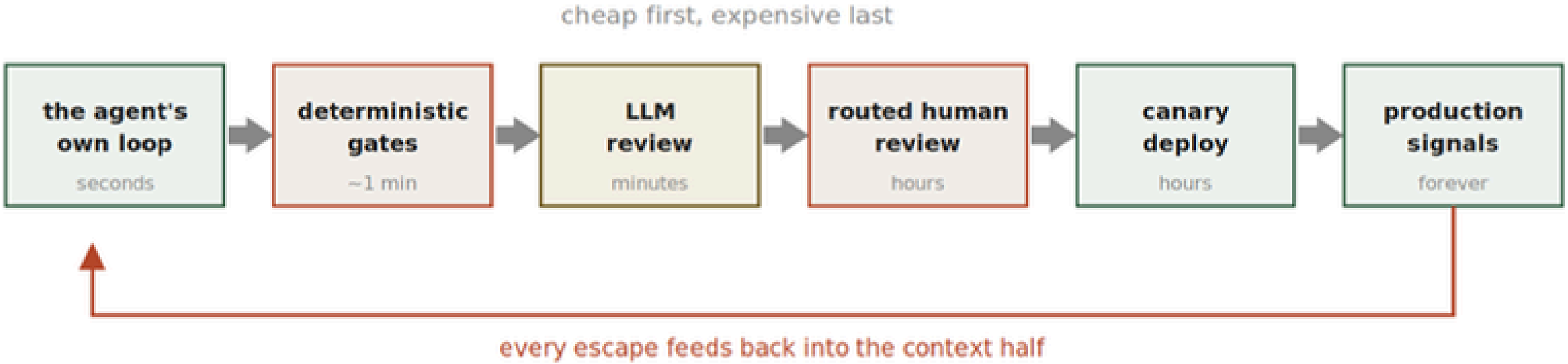
- More context helps, until it doesn't
- Effective context is often a fraction of the advertised window
- Quality degrades before the window is ever full
- The problem is not space, it is noise

Summarization

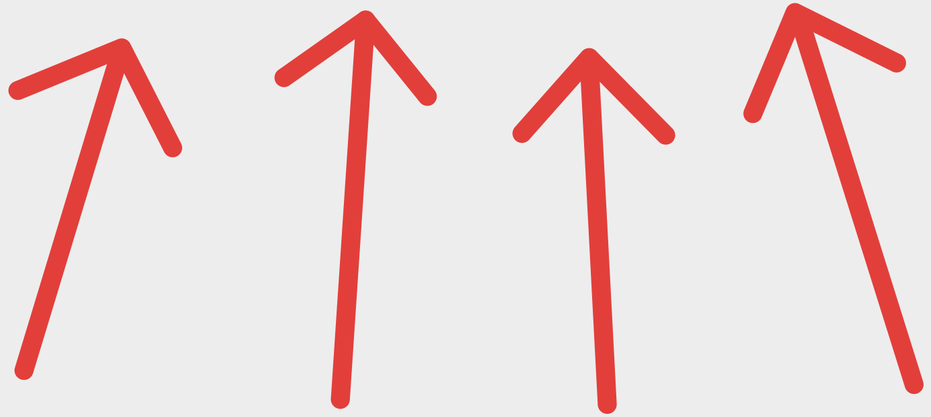



ARCHITECT
ARCHITECTURAL KNOWLEDGE
ADRs
DOMAIN / REASONING
Skills
"how to reason"
AGENTS.md
navigation layer
FEEDFORWARD
AI AGENT
CODE / DESIGN
FEEDBACK
tests
linters
fitness functions
semantic review
AGENT LOOP

Deterministic tools to the rescue



Shift left for the win



My philosophy

- “**Shift Left**” as much as possible
- Keeping **documentation up-to-date**
- Leverage **fitness functions**
- **Providing feedback to** your coding agents
- Reaching “**Fast Flow**”

	AGENTS.md	Rule	Skill
	always-loaded orientation	*a constraint the agent must respect*	*a workflow the agent can perform*
SCOPE	everything in this repository	one constraint, one glob it applies to	one kind of task, start to finish
LOADED	every session, every task	always, or when a matching path is touched	only when its description matches
SHAPE	orientation, commands, facts about the repo	the constraint, plus the reason it exists	steps, with references and scripts it can call

A fact about where you are goes in AGENTS.md.

A constraint that must always hold is a rule.

A procedure with steps is a skill.

What garbage context looks like

Sign	What goes wrong	Fix
A rule nobody can check	"Keep the code maintainable" cannot be enforced	"Use Decimal, never float, for money"
A don't without a do	The agent picks blindly among the options you left out	Name the repository layer when you forbid the ORM
A line added before a mistake happened	Every session steers toward a problem that may never occur	Add a rule after a real mistake
A file past 150 to 200 lines	Each new rule makes the others less likely to be followed	Link out to longer material
A rule without its reason	The agent applies it rigidly and nobody knows when to delete it	Write the reason next to the rule
A narrow rule loaded everywhere	It fires on tasks it has nothing to do with	Give it a glob

If garbage context leads to garbage decisions, who is responsible for the context



Thanks for listening

Luca Mezzalira

https://www.linkedin.com/in/lucamezzalira
https://x.com/lucamezzalira