Architectural Fitness in Practice:

Defining & Measuring your architecture goals with fitness functions

Rene Weiß

Consultant, Software & enterprise architect, pragmatic agile practitioner

Key focus areas

- Independent software / enterprise architecture consultant
- Software architecture reviews (ATAM, ..)
- End-to-end transformation consultant
  (from „C-Level to development teams and back“ 😉)

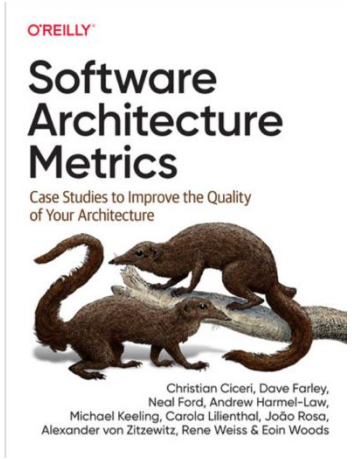
rene@rw-it.consulting

@renebianco

xing.to/rweiss

linkedin.com/in/renebianco

www.rw-it.consulting



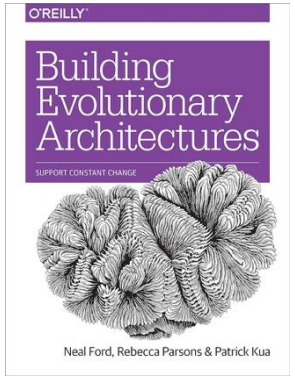
Co-Author
Software Architecture Metrics

“The Fitness Function Testing Pyramid: An Analogy for Architectural Tests and Metrics”

Rene Weiss IT Consulting

How did I start with “fitness functions”?

- **First client requests (2019):**
  - „We want to work have an evolutionary software architecture“ &
  - “Help us introduce fitness functions” – and no one knew what that really meant in practice.
- **I saw the same pattern everywhere:**
  - Architecture goals on slides (at best; or not even written down at all)
  - But nowhere: Concrete & measurable objectives or automated guardrails.
- **Foundation: Building evolutionary architectures (Ford, Parsons, Kua)**
  - The concepts in there were too high-level to work with (my) teams on fitness functions
  - I wanted something more concrete that teams can use in their work
  - A former colleague and I wrote an article about it



This talk distills what worked in real projects into a practical way to define and measure architecture goals.

Evolutionary architecture

An evolutionary architecture supports guided, incremental change across multiple dimensions.

Building Evolutionary Architectures (O'Reilly, 2017)

- **Architecture** must **evolve** as **requirements, scale,** and **constraints change.**
- **Key question:** How do we know if a change improves or degrades the architecture?

→ Fitness functions make architectural change measurable.

Fitness Function

Borrowed from evolutionary computing:
A **fitness function** is an objective function, used to summarize **how close** a given **design solution** is **to** achieving the **set aims**.

Building Evolutionary Architectures (O'Reilly, 2017)

- Think of it as **acceptance criteria** for your **architecture**
- It makes an **architecture concern explicit, measurable** and **verifiable** instead of just a slide or principle

Sample Fitness Function

Target

- Check if our test coverage is sufficient?

Implementation idea:

- Run integration tests
- Measure code coverage

Fitness Function(s):

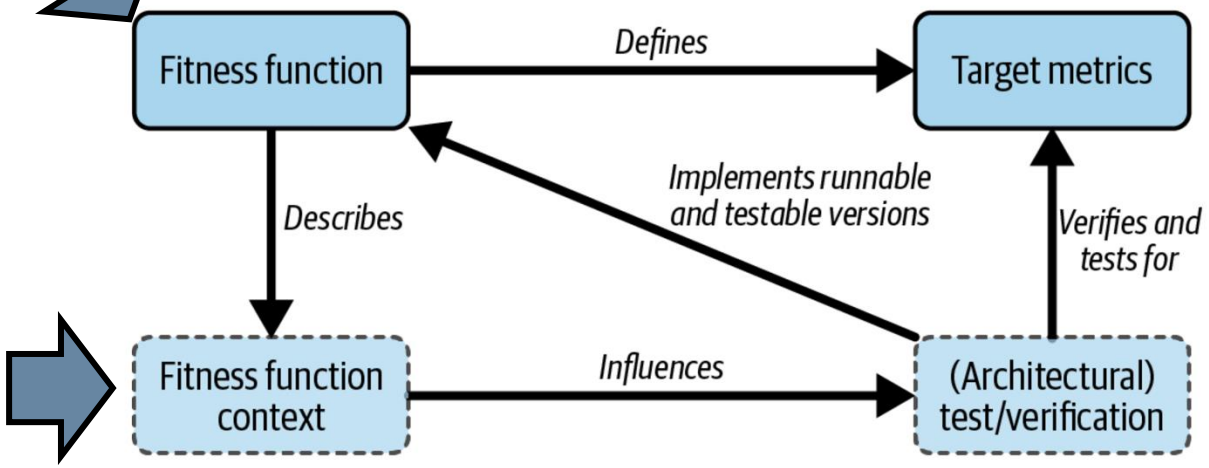
- Integration test code coverage > 0.6

When:

- Every nightly integration-test build

Fitness Function vs. Architectural test

Fitness Function(s):
- Integration test code coverage > 0.6
- + the context



Quality goal:
- Maintainability

Implementation idea:
- Run integration test
- Measure coverage

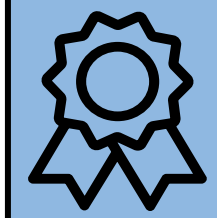
When:
- Every nightly integration-test build

Where:
- CI/CD pipeline

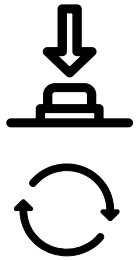
Automated verification of target metric

Fitness Function context

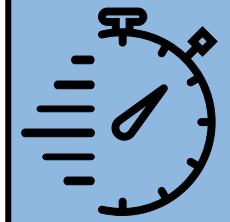
This is my **catalogue** of **building blocks** to design **a fitness function**:



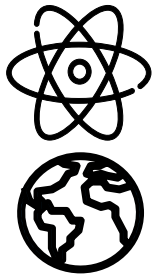
Quality goal
- Maintainability
- Reliability
- Performance
- …



Execution type
- Triggered
- Continuous

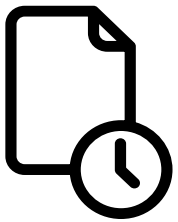


Monitoring type
- 0/1
- Value/Threshold
- Trend



Breadth of feedback
- Atomic
- Holistic

Execution location
- CI/CD
- Test environment
- Production



Fitness function lifecycle
- Temporary
- Permanent

Define your quality goals (= architecture goals)


Functional suitability


Performance


Compatibility


Usability


Reliability


Security


Maintain-ability


Portability

(ISO 25010)

+ Add a short, context-specific definition: what does this goal mean for your system?

Example: Maintainability

A small team can implement a typical feature (e.g., new business rule) without touching more than 2–3 modules, and without breaking existing customer journeys.
The architecture supports safe refactoring through automated tests and clear module boundaries. The layered architecture is never violated.

How I would start with fitness functions…

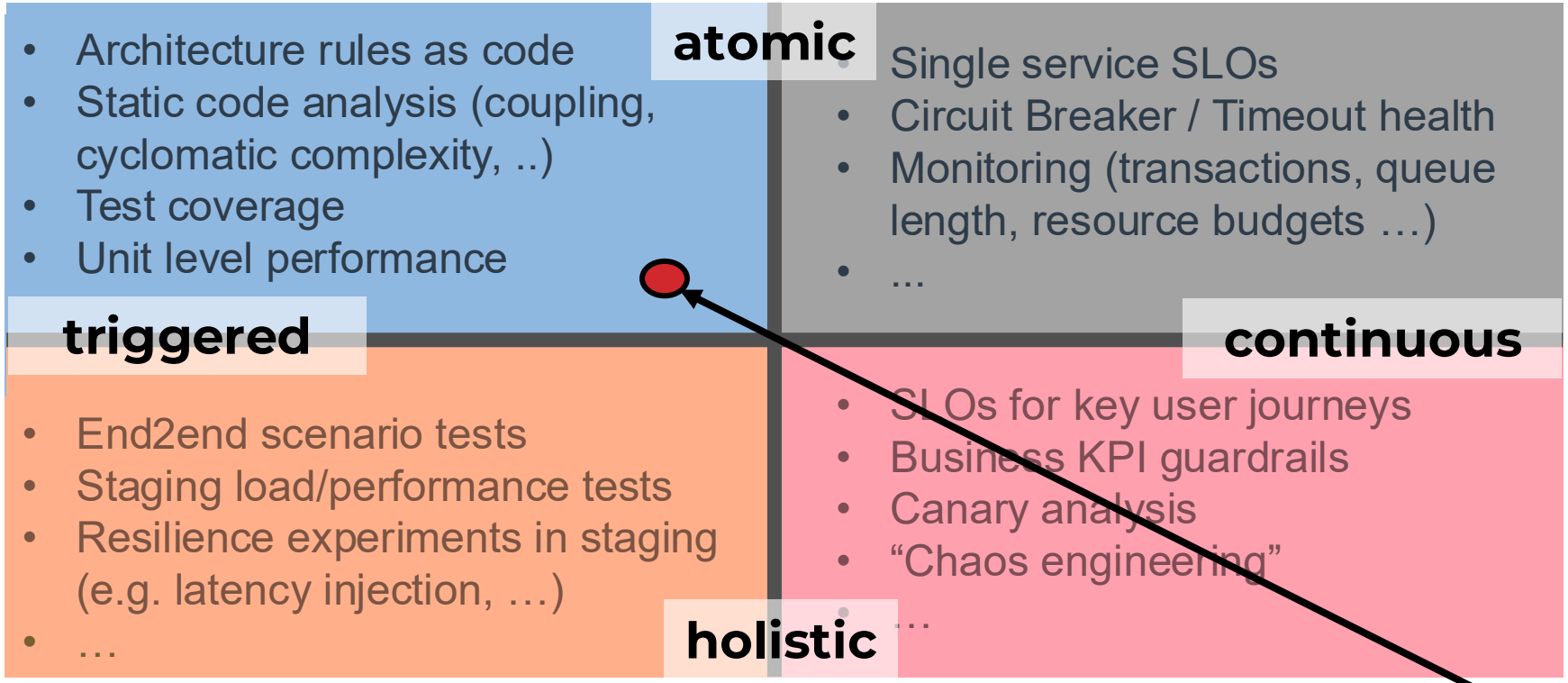
- Write down your **top 3 quality goals** (dedicated *.md file for each quality goal; put the file in your repo)
- Describe your **fitness functions and their context** (use the catalog!) and the **target metric** (again, in an *.md file)
- **Implement** a test that **verifies** the **target metric**

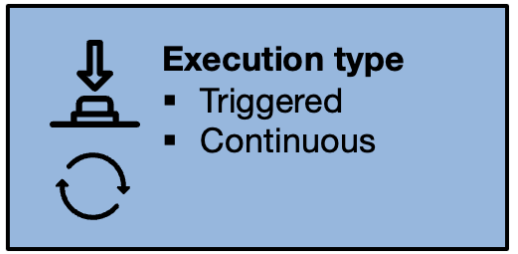
These *.md files become quite handy when working with tools like …

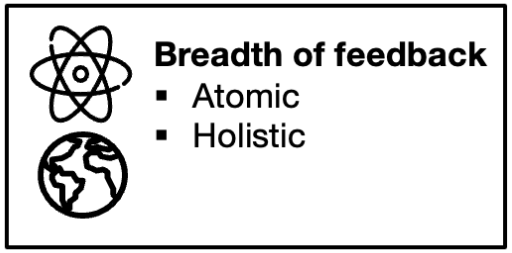
What this actually is:

- The fitness function is architectural documentation
- Architectural documentation => architecture communication
- An automated test with the verification is the “icing on the cake”

Fitness Function quadrants



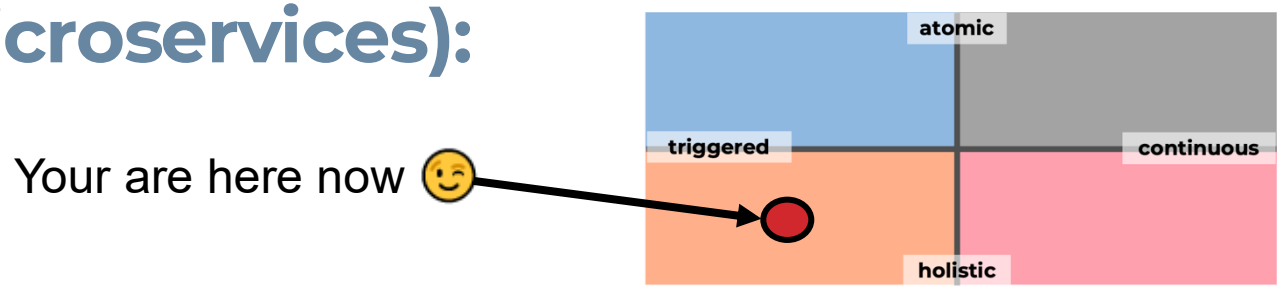




Fitness Function(s):

- Integration test code coverage > 0.6

Sample Fitness Function (Microservices): Resilience Under Latency



Architecture goal (Quality goal):

Reliability — Critical user journeys must remain functional when downstream services become slow.

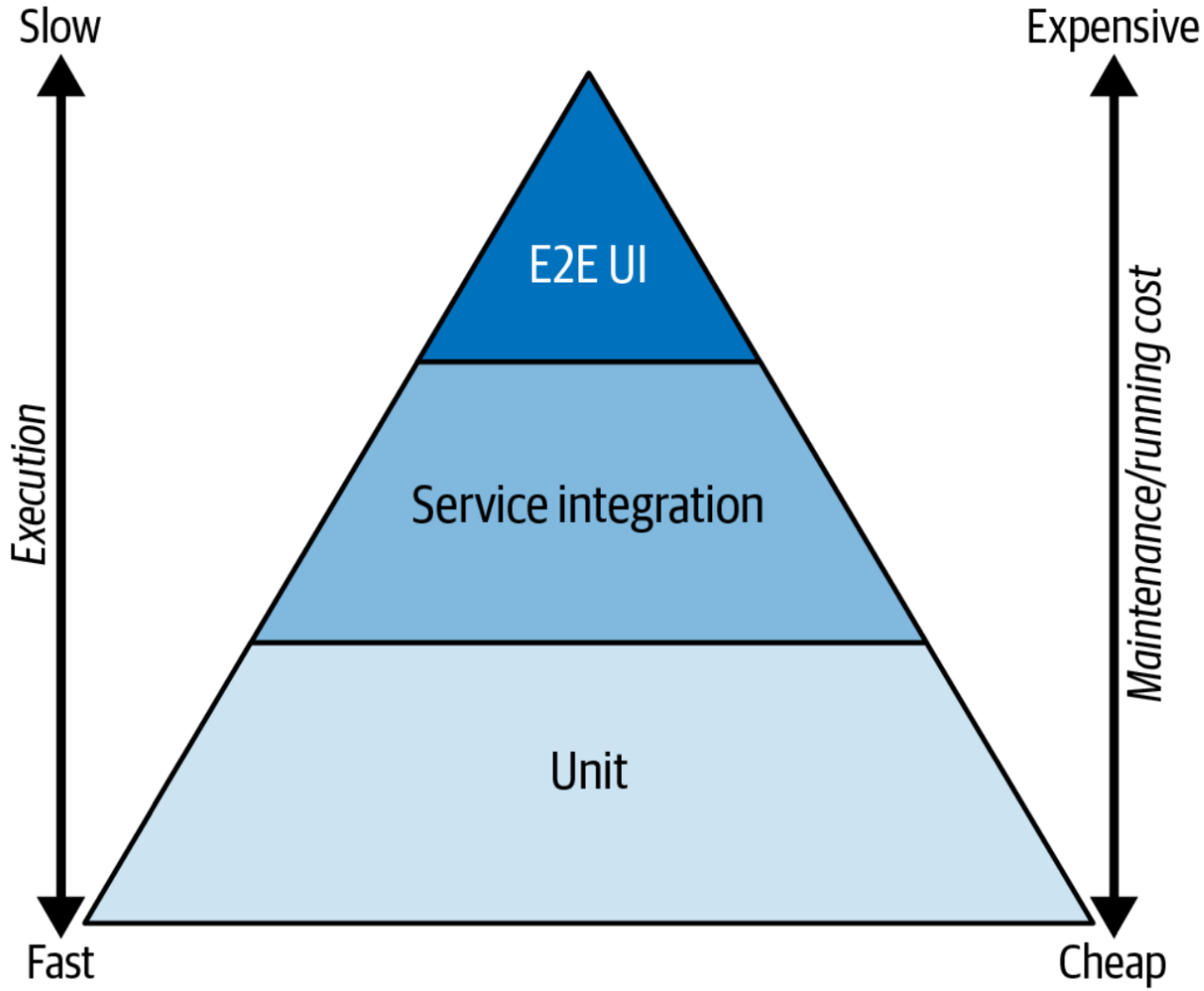
Fitness Function:

When Service B is degraded with +3000ms latency, the journey in Service A must still succeed without cascading failures.

Fitness Function Context:

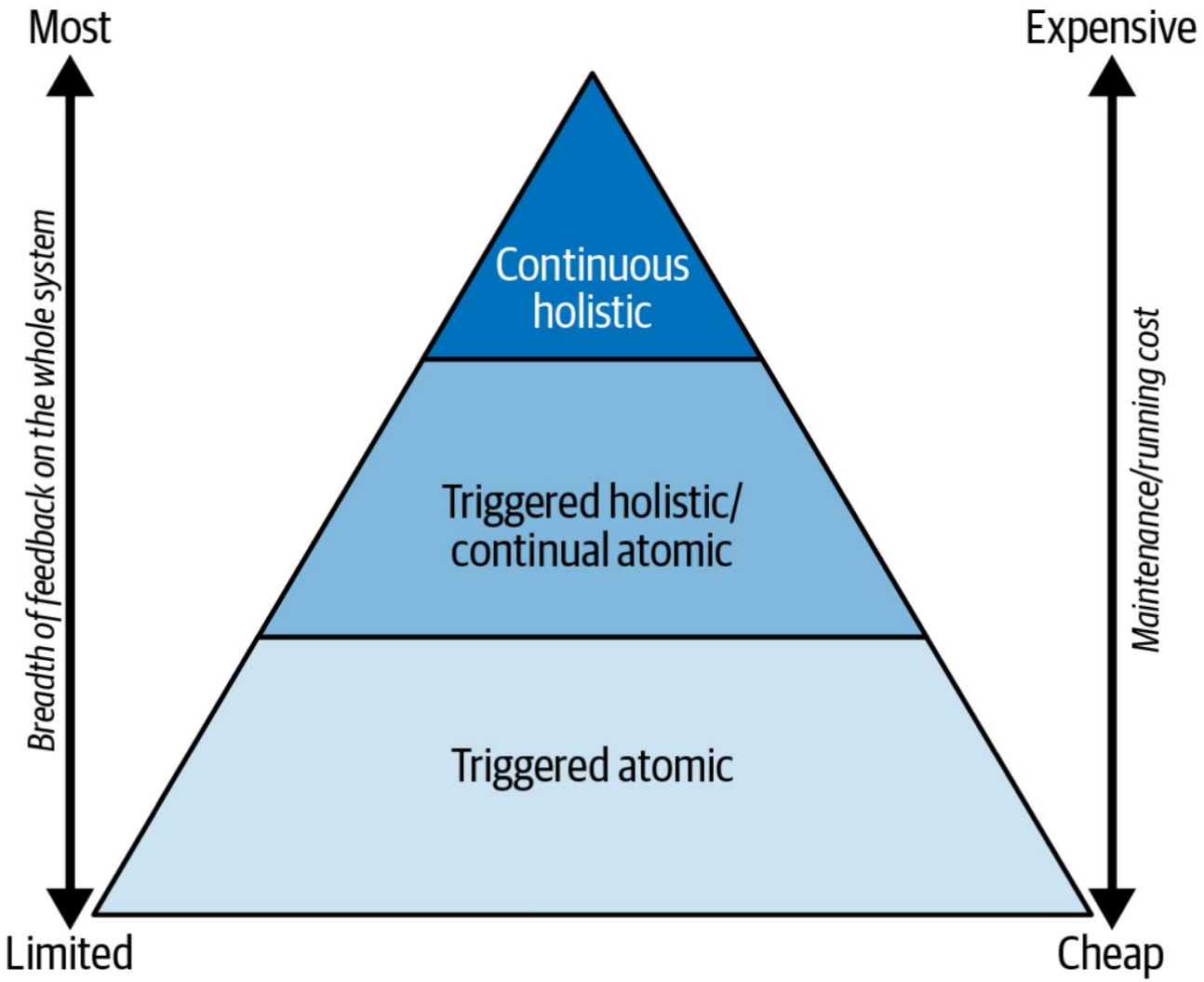
- Execution type: Triggered (nightly)
- Monitoring type: 0/1 (test doesn't fail)
- Breadth of feedback: Holistic (end-to-end journey)
- Execution location: Staging environment
- Test lifecycle: Permanent

Quick detour: The testing pyramid

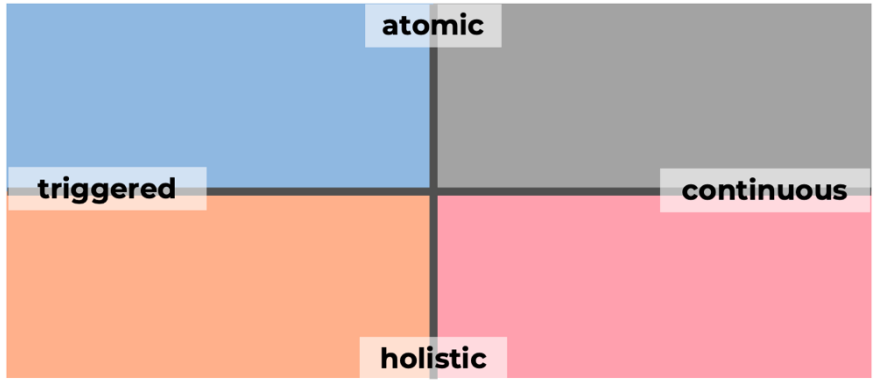


"...what does this have to do with fitness functions?"

The Fitness Function Testing pyramid



Fitness Function quadrants => mapped to the pyramid



Fitness Function Testing Pyramid – Example (Online Shop)

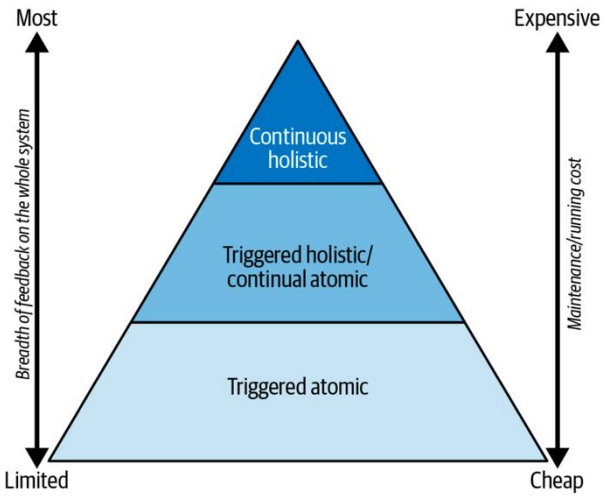
- **Bottom layer: Cheap, atomic fitness functions**
  - Service boundary rule: No UI layer directly calls the database; all access goes through service APIs. (Tooling hint: ArchUnit)
  - Dependency direction rule: checkout-service must not depend on catalog-service packages/modules (no compile-time coupling). (Tooling hint: ArchUnit)
  - API contract compatibility: All public API changes must be backward compatible (consumer-driven contract tests pass). (Tooling hint: Pact)
  - Performance test (unit-level): Price calculation completes within < 50ms for 95% of cases (local benchmark).
  - Resilience configuration guardrail: All outbound calls from checkout-service have timeouts + retries + circuit breaker configured (no “naked” HTTP calls).

- **Middle layer: Broader, scenario-based fitness functions**
  - Resilience under latency (key journey): Inject +500ms latency 2 downstream services → Checkout still succeeds (no timeouts/cascading failures).
  - End-to-end checkout SLO (staging): Full checkout flow is < 2500ms

- **Top layer: Holistic fitness function**
  - Continual monitoring of checkout rate: Checkout completion rate per hour must stay above 65% (rolling average). If it drops below threshold, trigger incident investigation

Recap – Fitness Function definition

- ❑ **Fitness functions make architecture goals explicit and measurable**
- ❑ **Architectural tests / automated checks validate the target metrics** in CI, test environments, or production.
- ❑ **Design fitness functions using the context catalog:** execution type, quality goal, monitoring type, breadth, location, lifecycle.
- ❑ **Choose the and define the quality goal first (e.g. ISO 25010)**
- ❑ **Pick the right level of feedback:** atomic vs. holistic, triggered vs. continuous
  → **Fitness Function Testing Pyramid.**



Where to use GenAI?!

You can use it as your assistant in every step…

Iterate as you learn!

- **Draft Quality Goals**
- **Draft Fitness Function**
- **Implement & maintain tests** based on the quality goals and fitness functions
- **Improve & refactor your codebase**

Just drafts!
You and the team (and the stakeholders) should decide on the final definitions!

This has become a lot easier now!! 😈

But YOU STILL OWN the changes and your code!

GenAI flow 1/3

Quality goals – draft quality goals for my software

full prompt at the end of slide-deck

Claude Code v2.1.19
Opus 4.5 · Claude Pro
~/Documents/Development/broker_document_extractor

```
❯ You are Claude Code acting as a senior software/enterprise architect.

  Goal
  - Define the top 3 architectural quality goals for THIS solution, using the 8 ISO/IEC 25010 product quality characteristics as the candidate
  set.
  - Write them to a dedicated file: quality_goals.md
  - Add a link to that file from claude.md

  ISO 25010 candidate categories (use these labels)
  1) Functional suitability
  2) Performance efficiency
  3) Compatibility
  4) Usability
  5) Reliability
  6) Security
  7) Maintainability
  8) Portability

  Process (do not skip)
  1) Understand the solution context from the repository
  - Read claude.md, README, docs/, package manifests, in
  - Identify: primary users, deployment model, key data types (especially sensitive), availability expectations, operational complexity,
  and expected change rate.
  2) Select the TOP 3 ISO 25010 characteristics that are most architecturally important for this solution.
  - Do not pick “Functional suitability” unless there is a strong reason; architectural quality goals should drive structural decisions
  beyond feature completeness.
  - Avoid “everything is important.” Force trade-offs: pick only 3.
```

Output (=> manually review/rework before proceeding):

docs > architecture > quality_goals.md > # Architectural Quality Goals

```
1  # Architectural Quality Goals
2
3  This document defines the top 3 quality characteristics (per ISO/IEC 25010) driving architectural decisions for the Broker Document Extractor.
4
5  ---
6
7  ## Security
8
9  **Meaning:** All broker document content, extracted text, ISINs, amounts, and transaction history must remain on the user's local machine. The
   system must never transmit sensitive financial data to external services. This is not optional; it is a hard constraint for a tool processing
   personal investment records.
10
11 **Implications for our architecture:**
12 - Network isolation: The only permitted HTTP target is `localhost:11434` (Ollama). All other outbound connections are prohibited.
13 - No cloud storage, no telemetry, no external logging services.
14 - File operations confined to local directories (`data/uploads/`, SQLite database).
15 - Default DRY-RUN mode prevents unintended file mutations.
```

GenAI „Inception“ 😄: I used ChatGPT to create the prompt above and manually tweaked it

GenAI Flow 2/3

Fitness Functions – create draft fitness functions for “maintainability”

```
❯ You are Claude Code acting as a senior software/enterprise architect.

  Goal
  Create two concrete, sample architecture fitness functions for the ISO 25010 quality goal: Maintainability, based on our existing documentation
  (./docs/architecture/quality_goals.md)

  Inputs
  - Read and use: ./docs/architecture/quality_goals.md
  - Read and update: ./claude.md

  Task A — Create file
  Create/overwrite: ./docs/architecture/fitness_functions_maintainability.md

  File requirements
  - Title: “Fitness Functions — Maintainability”
  - Include exactly 2 fitness functions.
  - They must be concrete and automatable in a typical engineering setup (CI/CD
  - Each must clearly map back to the Maintainability quality goal in ./docs/arc
  - Keep it concise and practical; no filler.

  Each fitness function MUST be formatted like this:

  ## <Short fitness function name>
  **Description:** <1–3 sentences>
  **Target metric:** <specific metric and target, e.g., thresholds, budgets, tre

  **Context**
  - Execution type: <Triggered|Continuous>
  - Monitoring type: <0/1|Value/Threshold|Trend>
  - Breadth of feedback: <Atomic|Holistic>
  - Execution location: <CI/CD|Test environment|Production>
  - Fitness function lifecycle: <Temporary|Permanent>
  - Quality goal reference: <exact maintainability goal heading from quality_goa
```

full prompt at the end of slide-deck

Output (=> manually review/rework before proceeding):

docs > architecture > fitness_functions_maintainability.md > # Fitness Functions — Maintainability

```
1  # Fitness Functions — Maintainability
2
3  These fitness functions provide objective evidence that the system meets the **Maintainability** quality goal
   defined in [quality_goals.md](./quality_goals.md).
4
5  ---
6
7  ## File Size Budget
8
9  **Description:** Enforces the architectural constraint that source files remain small and focused (<200 lines).
   Large files increase cognitive load and make safe, incremental changes harder.
10
11 **Target metric:** All TypeScript source files in `packages/` must have ≤200 lines of code. Threshold: 0
   violations.
12
13 **Context**
14 - Execution type: Triggered
15 - Monitoring type: 0/1
16 - Breadth of feedback: Atomic
17 - Execution location: CI/CD
18 - Fitness function lifecycle: Permanent
19 - Quality goal reference: Maintainability
```

GenAI Flow 3/3

Implement first tests

full prompt at the end of slide-deck

```
❯ You are Claude Code acting as a senior TypeScript engineer and guardian of the architecture-governance.

  Goal
  Add an executable "fitness function test" run target using pnpm, and implement maintainability fitness function tests in a dedicated test file.

  Repository analysis (do not skip)
  1) Inspect the repo to determine:
  - Package manager usage (pnpm workspace), Node version constraints, TypeScript config
  - Test framework in use (Vitest), test folder conventions, and current scripts in package.json
  2) Locate the fitness function definition file:
  - ./docs/architecture/fitness_functions_maintainability.md
  - Read it and identify the two fitness functions, their target metrics, and how they should be verified.
  3) Determine what is realistically automatable using repo artifacts (source tree, tsconfig, eslint config, coverage outputs, dependency graph tools, etc.).
  - Do not invent tools. Use what is already present in devDependencies and CI scripts.
  - If a needed capability is missing, implement the check with lightweight TypeScript code (filesystem traversal, ts-morph, simple parsers) or add a minimal dependency
  ONLY if necessary and justified.

  Implementation tasks

  Task A — Add a dedicated pnpm run target
  1) Update package.json (and workspace package.json if applicable) to add a script that runs ONLY the fitness function tests.
  - Name the script clearly, e.g.:
  - "test:fitness" (preferred)
  - It must not run the full test suite.
  - It must be compatible with the test runner already used in the repo.
  - It must target ONLY tests under: /tests/fitness_function/**

  2) If the test runner supports patterns/grep, configure it so:
  - `pnpm test:fitness` executes only the fitness function tests.
  - Existing `pnpm test` remains unchanged.
```

```
● renew@RenesM4Workbook broker_document_extractor % pnpm --filter backend test:fitness

> backend@1.0.0 test:fitness /Users/renew/Documents/Development/broker_document_extractor/packages/backend
> vitest run tests/fitness_function


 RUN  v1.6.1 /Users/renew/Documents/Development/broker_document_extractor/packages/backend

 ✓ tests/fitness_function/maintainability.test.ts (2)
   ✓ Maintainability Fitness Functions (2)
     ✓ File Size Budget (1)
       ✓ all production source files must be ≤200 lines
     ✓ Layered Architecture Compliance (1)
       ✓ layers must not import from forbidden layers (routes → services → db)

 Test Files  1 passed (1)
      Tests  2 passed (2)
   Start at  21:13:05
   Duration  125ms (transform 14ms, setup 0ms, collect 13ms, tests 2ms, environment 0ms, prepare 30ms)
```

Output – automated tests:

quality_goals.md (Index) | fitness_functions_maintainability.md | maintainability.test.ts

packages > backend > tests > fitness_function > maintainability.test.ts > ...

```ts
/**
 * Maintainability Fitness Function Tests
 *
 * These tests provide automated evidence that the codebase meets the
 * Maintainability quality goal defined in:
 * ./docs/architecture/quality_goals.md
 *
 * Fitness functions defined in:
 * ./docs/architecture/fitness_functions_maintainability.md
 */

import { describe, it, expect } from 'vitest';
import { readFileSync, readdirSync, statSync } from 'fs';
import { join, relative } from 'path';

// Project root is 4 levels up from this test file
const PROJECT_ROOT = join(__dirname, '..', '..', '..', '..');

```

<-- Test result 💪

Key takeaways

- **Start with a clear quality goal, then design the fitness function using the catalog** (execution type, execution location, monitoring type, lifecycle)
- **Apply the Fitness Function Testing Pyramid**
  - Cheap, atomic checks at the bottom for fast feedback
  - Broader, scenario-based checks in the middle (cross-service / end-to-end)
  - A few holistic checks at the top (SLOs / key journeys)
- **Use GenAI to draft goals and propose candidate fitness functions—then validate, refine, and own the decision.**

Fitness functions won't replace architecture work—but they make architecture goals testable and continuously enforceable

Tomorrow: Pick one system, define 1-2 goals, formulate 2 fitness functions and automate a test → start at the bottom of the pyramid

THANK YOU!

rene@rw-it.consulting

@renebianco

xing.to/rweiss

linkedin.com/in/renebianco

www.rw-it.consulting

Resources

Claude prompt to draft quality goals

```
You are Claude Code acting as a senior software/enterprise architect.

Goal
- Define the top 3 architectural quality goals for THIS solution, using the 8 ISO/IEC 25010 product quality characteristics as the candidate set.
- Write them to a dedicated file: quality_goals.md
- Add a link to that file from claude.md

ISO 25010 candidate categories (use these labels)
1) Functional suitability
2) Performance efficiency
3) Compatibility
4) Usability
5) Reliability
6) Security
7) Maintainability
8) Portability

Process (do not skip)
1) Understand the solution context from the repository:
   - Read claude.md, README, docs/, package manifests, infra/deploy files, and skim key modules.
   - Identify: primary users, deployment model, key data types (especially sensitive), availability expectations, operational complexity, and expected change rate.
2) Select the TOP 3 ISO 25010 characteristics that are most architecturally important for this solution.
   - Do not pick "Functional suitability" unless there is a strong reason; architectural quality goals should drive structural decisions beyond feature completeness.
   - Avoid "everything is important." Force trade-offs: pick only 3.
3) For each selected goal, write:
   - A short plain-language meaning (2-4 sentences) tailored to this solution (not a generic textbook definition).
   - 2-4 concrete implications / architectural strategies (bullets). Keep them specific (e.g., "idempotent handlers," "zero-trust service-to-service auth," "SLO-driven error budgets," "bounded contexts," etc.) based on what you observed in the repo.
   - 2-3 lightweight acceptance signals (how we know we're meeting it), ideally measurable (SLOs, test coverage type, latency budgets, RTO/RPO, severity targets, etc.). If exact numbers aren't knowable from repo context, propose reasonable defaults and label them as "initial targets."

Output requirements
A) Create/overwrite file: quality_goals.md (repo root unless claude.md implies another docs location).
   - Title: "Architectural Quality Goals"
   - One section per goal:
     - "## <ISO25010 characteristic name>"
     - "Meaning"
     - "Implications for our architecture"
   - Keep the entire file concise: ~10-15 lines max per goal

B) Update claude.md
   - Add a clearly visible link to quality_goals.md using a relative markdown link: [Architectural Quality Goals](./docs/architecture/quality_goals.md)
   - Place it in the most appropriate existing section (e.g., Documentation, Architecture, Engineering Guidelines). If no suitable section exists, create a small "Documentation" or "Architecture" section without disrupting the rest of the file.
   - Do not remove or rewrite unrelated content.

Constraints
- Be opinionated and consistent with the repo's reality. If the solution handles sensitive data, Security should likely be in the top 3; if it's a platform component, Maintainability/Reliability often dominate; if it's user-facing and adoption-driven, Usability may matter.
- Do not invent domain facts. If something is uncertain, state the assumption briefly and keep going.
- Ensure both files format cleanly in Markdown.

Deliverables checklist (must satisfy)
- [ ] quality_goals.md exists with exactly 3 goals chosen from ISO 25010 and includes tailored meaning + implications + acceptance signals
- [ ] claude.md contains a working link to ./docs/architecture/quality_goals.md
- [ ] No unrelated edits outside these documentation changes
```

Resources

Claude prompt to define 2 fitness functions

```
You are Claude Code acting as a senior software/enterprise architect.

Goal
Create two concrete, sample architecture fitness functions for the ISO 25010 quality goal:
Maintainability, based on our existing documentation (./docs/architecture/quality_goals.md)

Inputs
- Read and use: ./docs/architecture/quality_goals.md
- Read and update: ./claude.md

Task A — Create file
Create/overwrite: ./docs/architecture/fitness_functions_maintainability.md

File requirements
- Title: "Fitness Functions — Maintainability"
- Include exactly 2 fitness functions.
- They must be concrete and automatable in a typical engineering setup (CI/CD and/or production),
with measurable targets.
- Each must clearly map back to the Maintainability quality goal in
./docs/architecture/quality_goals.md.
- Keep it concise and practical; no filler.

Each fitness function MUST be formatted like this:

## <Short fitness function name>
**Description:** <1-3 sentences>
**Target metric:** <specific metric and target, e.g., thresholds, budgets, trend direction>

**Context**
- Execution type: <Triggered|Continuous>
- Monitoring type: <0/1|Value/Threshold|Trend>
- Breadth of feedback: <Atomic|Holistic>
- Execution location: <CI/CD|Test environment|Production>
- Fitness function lifecycle: <Temporary|Permanent>
- Quality goal reference: <exact maintainability goal heading from quality_goals.md>

**How to implement (draft and concise; maximum 10 lines):**
- <2-3 bullets describing how it would be automated, what tool signals/artifacts it uses (e.g.,
linting, complexity, architecture tests, code ownership, dependency graphs, test coverage on changed
files, etc.)>
- <Be specific, but do not invent repo-specific tools if not present; if uncertain, state a
reasonable assumption explicitly.>

Task B — Update claude.md
1) Add a link to the new file using a relative Markdown link:
   [Fitness Functions — Maintainability](./docs/architecture/fitness_functions_maintainability.md)
2) Place the section and link in the most appropriate existing location (Architecture / ...).

Constraints
- Do not modify unrelated content.
- Do not guess the content of quality_goals.md: read it and reference the exact maintainability
wording/heading.
- Ensure links are correct and render in Markdown.

Deliverables checklist
- [ ] ./docs/architecture/fitness_functions_maintainability.md created with exactly 2 maintainability
fitness functions
- [ ] ./claude.md updated with: link to the maintainability fitness function file
- [ ] No unrelated edits
```

Resources

Claude prompt to implement 2 tests for fitness functions

```
You are Claude Code acting as a senior TypeScript engineer and guardian of the architecture-
governance.

Goal
Add an executable “fitness function test” run target using pnpm, and implement maintainability
fitness function tests in a dedicated test file.

Repository analysis (do not skip)
1) Inspect the repo to determine:
   - Package manager usage (pnpm workspace), Node version constraints, TypeScript config
   - Test framework in use (Vitest), test folder conventions, and current scripts in package.json
2) Locate the fitness function definition file:
   - ./docs/architecture/fitness_functions_maintainability.md
   - Read it and identify the two fitness functions, their target metrics, and how they should be
verified.
3) Determine what is realistically automatable using repo artifacts (source tree, tsconfig, eslint
config, coverage outputs, dependency graph tools, etc.).
   - Do not invent tools. Use what is already present in devDependencies and CI scripts.
   - If a needed capability is missing, implement the check with lightweight TypeScript code
(filesystem traversal, ts-morph, simple parsers) or add a minimal dependency ONLY if necessary and
justified.

Implementation tasks

Task A — Add a dedicated pnpm run target
1) Update package.json (and workspace package.json if applicable) to add a script that runs ONLY the
fitness function tests.
   - Name the script clearly, e.g.:
     - "test:fitness" (preferred)
   - It must not run the full test suite.
   - It must be compatible with the test runner already used in the repo.
   - It must target ONLY tests under: /tests/fitness_function/**

2) If the test runner supports patterns/grep, configure it so:
   - `pnpm test:fitness` executes only the fitness function tests.
   - Existing `pnpm test` remains unchanged.

Task B — Implement maintainability fitness function tests
Create the file:
- /tests/fitness_function/maintainability.test.ts

This file must:
1) Implement tests that correspond to the maintainability fitness functions defined in:
   - ./docs/architecture/fitness_functions_maintainability.md

2) For each fitness function, encode:
   - The measurement logic (how to compute the metric)
   - The assertion against the target threshold/goal
   - Helpful failure messages that explain what violated the fitness function and how to remediate

3) The tests must be deterministic and fast:
   - No network access
   - No reliance on production systems
   - Avoid heavyweight whole-repo compilation unless already the norm

Task C — Documentation update
Update ./claude.md to include:
- How to run the fitness function tests:
  - `pnpm test:fitness`
- Where they live:
  - `/tests/fitness_function/`
- A short note that fitness function tests are intended to provide automated evidence for
architectural quality goals.

Constraints
- Do not change unrelated behavior of the existing test suite.
- Do not add large new dependencies without strong justification.
- Keep changes minimal and idiomatic to the existing repo conventions.
- Ensure all added/modified files are formatted consistently with the repo.

Deliverables checklist
- [ ] package.json updated with `test:fitness` (or equivalent) that runs only
/tests/fitness_function/**
- [ ] /tests/fitness_function/maintainability.test.ts implemented to verify the maintainability
fitness functions defined in ./docs/architecture/fitness_functions_maintainability.md
- [ ] ./claude.md updated with instructions for running fitness function tests
- [ ] Existing test commands still work as before
```

Resources

My claude.md with quality goals.md & fitness function catalog definition

```
## Architecture
- [Architectural Quality Goals](./docs/architecture/quality_goals.md) - Top 3 ISO 25010 quality characteristics driving design decisions


### Fitness Function Definition
A fitness function is an executable (or continuously evaluated) check that provides objective evidence a system is meeting an architectural quality goal. It defines what to measure and what "good" looks like, and it is designed to be automated where possible.


**Fitness function fields** (must be used for each function):
1. **Fitness function description** - What the function checks
2. **Target metric** - What is measured and the threshold/target to verify
3. **Fitness function context**:
- **Execution type**: Triggered or Continuous
- **Monitoring type**: 0/1, Value/Threshold, Trend
- **Breadth of feedback**: Atomic or Holistic
- **Execution location**: CI/CD, Test environment, Production
- **Fitness function lifecycle**: Temporary or Permanent
- **Quality goal reference**: Must reference a quality goal from [quality_goals.md](./docs/architecture/quality_goals.md) (use the exact heading/name)
```

Resources

My fitness_functions_maintainability.md

```
# Fitness Functions — Maintainability

These fitness functions provide objective evidence that the system meets the
**Maintainability** quality goal defined in
[quality_goals.md](./quality_goals.md).

---

## File Size Budget

**Description:** Enforces the architectural constraint that source files remain
small and focused (<200 lines). Large files increase cognitive load and make
safe, incremental changes harder.

**Target metric:** All TypeScript source files in `packages/` must have ≤200
lines of code. Threshold: 0 violations.

**Context**
- Execution type: Triggered
- Monitoring type: 0/1
- Breadth of feedback: Atomic
- Execution location: CI/CD
- Fitness function lifecycle: Permanent
- Quality goal reference: Maintainability

**How to implement:**
- Run a script that counts lines per `.ts` file in `packages/backend/src/` and
`packages/frontend/src/`, excluding test files and type declarations.
- Fail CI if any file exceeds 200 lines.
- Example: `find packages -name "*.ts" ! -name "*.test.ts" ! -name "*.d.ts" -
exec wc -l {} + | awk '$1 > 200 {print; exit 1}'`

---

## Layered Architecture Compliance

**Description:** Enforces unidirectional dependency flow between architectural
layers: `routes → services → db`. Reverse imports (e.g., a service importing
from routes, or db importing from services) break layer isolation, making
modules harder to test independently and increasing the risk of cascading
changes.

**Target metric:** Zero reverse-layer imports in backend source files.
Threshold: 0 violations.

**Forbidden import directions:**
- `services/` must NOT import from `routes/`
- `db/` must NOT import from `routes/` or `services/`
- `middleware/` must NOT import from `routes/`

**Context**
- Execution type: Triggered
- Monitoring type: 0/1
- Breadth of feedback: Atomic
- Execution location: CI/CD
- Fitness function lifecycle: Permanent
- Quality goal reference: Maintainability

**How to implement:**
- Scan import statements in each layer directory and check for forbidden
patterns.
- Fail CI if any reverse-layer import is found.
- Example:
  ```bash
  # Services must not import from routes
  grep -rE "from ['\"].*routes" packages/backend/src/services/ && exit 1
  # DB must not import from routes or services
  grep -rE "from ['\"].*routes|from ['\"].*services" packages/backend/src/db/ &&
exit 1
  # Middleware must not import from routes
  grep -rE "from ['\"].*routes" packages/backend/src/middleware/ && exit 1
  exit 0
  ```
```