Cognitive Load in Code

Why Simplicity Wins in System Design

About Me

**Director
Consulting**

CGI

Boston, MA

- 25 years of enterprise software development experience – from mainframes to Cloud-native and AI.

- Focused on application modernization and digital transformation for Fortune 500 customers.

- Drive architectural vision while staying deeply engaged with code and delivery.

Let's connect!

What are we going to talk about today?

🧠 **The Developer's Brain**

Understanding cognitive load and why it matters in software design

🕳️ **Where Complexity Hides**

Anti-patterns and subtle traps in code and architecture

🧩 **Designing for Simplicity**

Practical principles to reduce cognitive load in your systems

🛠️ **Tools and Techniques**

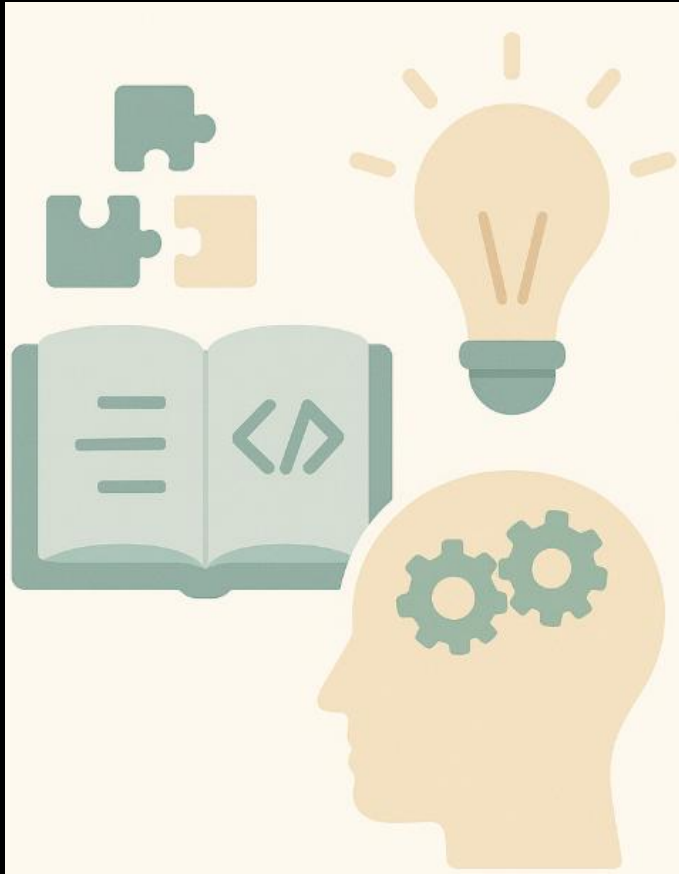IDE tips, team conventions, and documentation strategies that help

📚 **Real-World Stories**

Your experiences with encountering and solving design and code complexity

🎁 **Wrap-Up & Takeaways**

Patterns, tools, and habits to design systems that developers love

"Simplicity is the ultimate sophistication."

— Leonardo da Vinci

"Everything should be made as simple as possible, but not simpler."

— Albert Einstein

**"Programs must be written for people to read, and only incidentally for machines to execute."**

— Harold Abelson & Gerald Jay Sussman, Structure and Interpretation of Computer Programs

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
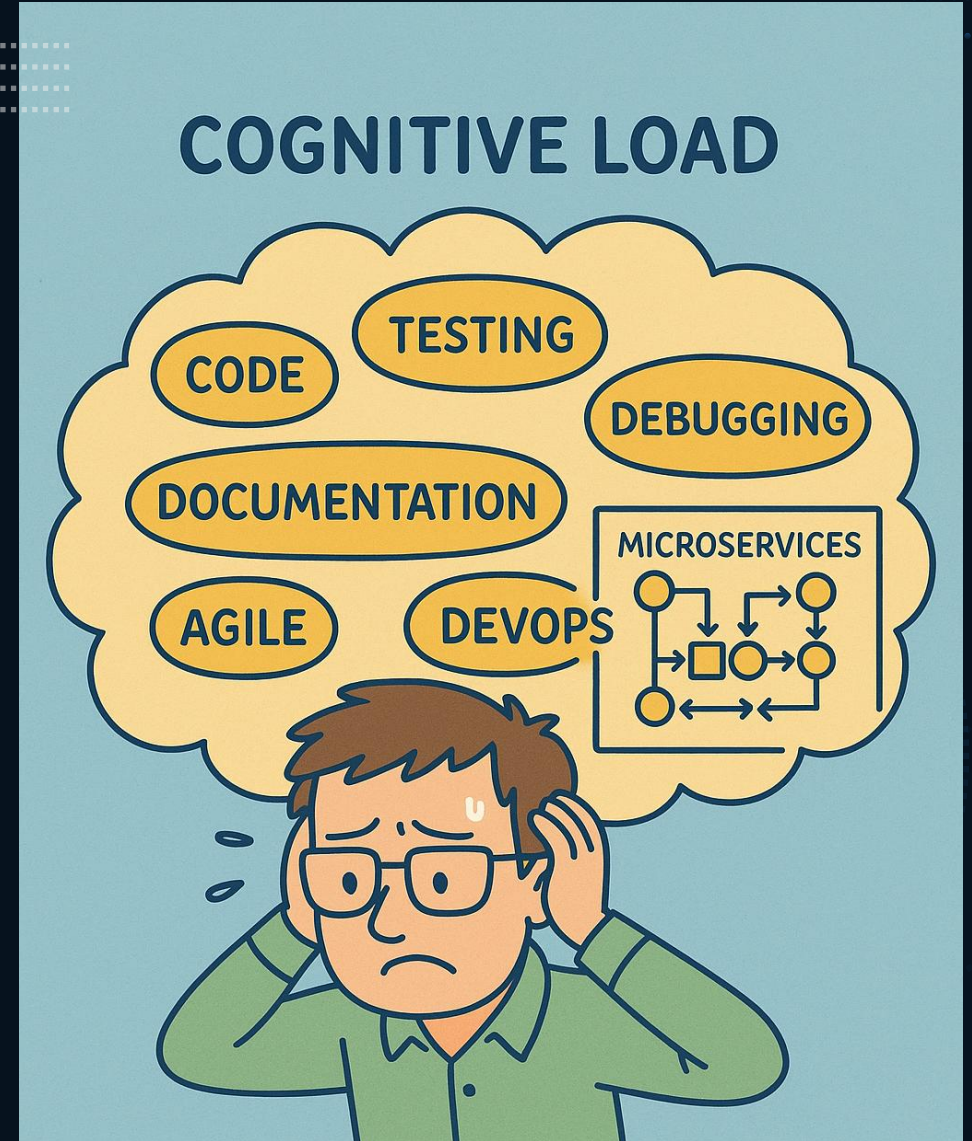
— Martin Fowler

The Developer's Brain

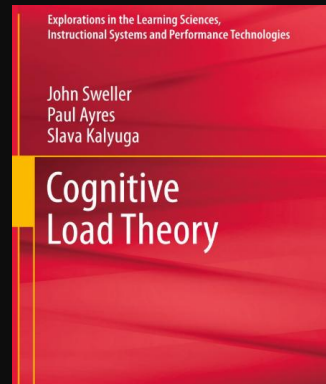Cognitive load and why it matters in software design
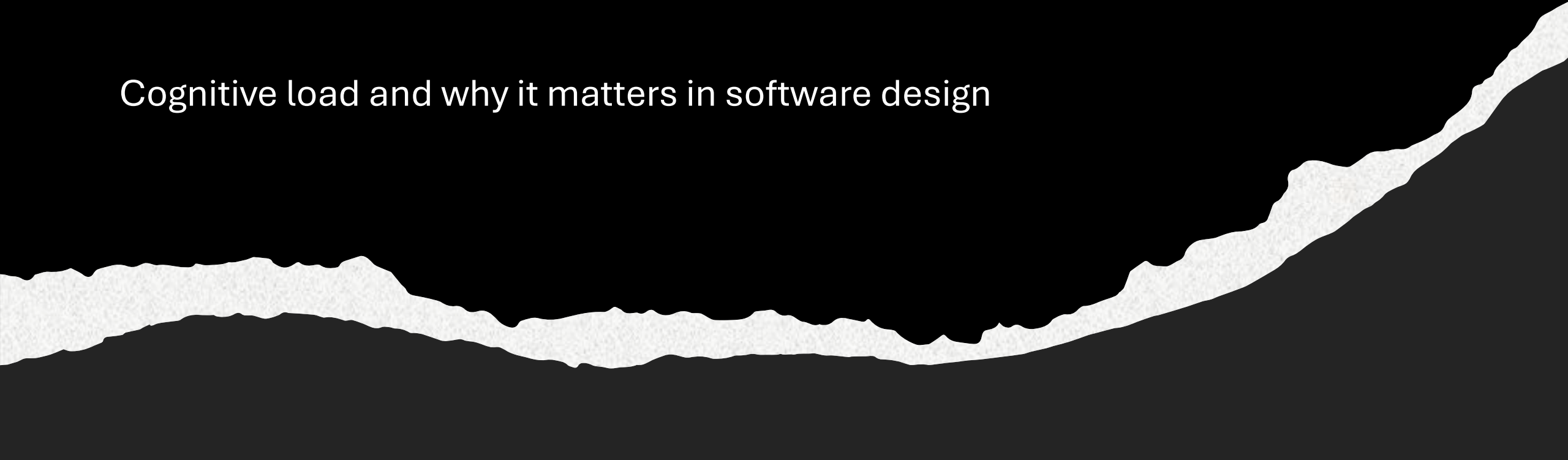
Does this look familiar?

For some, AI has added to the overload!

Cognitive Load Theory

- First formulated by psychologist John Sweller in 1988 during his research on problem-solving and learning.

- The goal of CLT was to develop teaching techniques that align with how our brains work.

- Built on two fundamental observations about human memory:

  - **Working Memory is Limited**: The average person's conscious mind can manage only about five to nine discrete pieces of information at a time (Miller's Law).

  - **Long-Term Memory is (Practically) Unlimited**: We can accumulate a vast repository of knowledge and experiences over time.

Examples of how working memory limitations challenge software developers

- Reading unfamiliar code requires holding new variables, APIs, and logic in working memory, often leading to cognitive overload.

- Debugging complex code is difficult due to the need to mentally track execution paths and multiple interdependent elements.

- Learning a new language demands retaining unfamiliar syntax and structures, which can quickly exceed short-term memory capacity.

- Reviewing large volumes of code at once can overwhelm working memory, reducing the quality of feedback and issue detection.

- Navigating complex APIs with many functions and parameters can strain memory, making it harder to recall essential details.

Types of Cognitive Load

Intrinsic Load	Germane Load	Extraneous Load
• Corresponds to the **essential complexity** of the problem we're solving. • For example, implementing a complex algorithm (say, cryptographic encryption) or understanding a challenging domain (like multi-threaded concurrency or a highly mathematical business logic) has high intrinsic load.	• Mental effort devoted to learning or developing deeper understanding of new material. **Productive cognitive effort**. • For example, the effort a developer spends to really comprehend the system design or to grok a new framework's concepts.	• Mental overhead that we inadvertently dump on others (or ourselves) on top of the intrinsic complexity. **Avoidable complexity**. • For example, poorly named variables, deeply nested logic, inconsistent coding styles, irrelevant details, or "clever" tricks that obscure intent

Unavoidable

Beneficial

Unnecessary

Spot the items causing extraneous cognitive load in this code

```java
public class P {

    public static void x(String[] a) {
        String q = "John";
        int[] z = { 100, 200, 150 };
        int r = 0;

        for (int i = 0; i < z.length; i++) {
            if (z[i] > 150) {
                r += z[i];
            }
        }

        System.out.println("Name: " + q);
        System.out.println("Result: " + r);
    }
}
```

1. **Poor Naming**:

   - P, x, a, q, z, r — these names convey no meaning.

   - The reader must infer the purpose of every variable and method, increasing mental effort.

2. **Lack of Abstraction**:

   - The logic is embedded directly inside the main-like method with no explanation or structure.

3. **Hardcoded Data**:

   - Values like 100, 200, 150 are not explained or named — increasing ambiguity.

4. **No Comments or Context**:

   - There's no explanation of what the program is trying to do.

Here's an improved version, with far less cognitive load

```java
public class SalaryProcessor {

    public static void main(String[] args) {
        String employeeName = "John";
        int[] monthlyBonuses = { 100, 200, 150 };
        int totalHighBonuses = calculateHighBonuses(monthlyBonuses, 150);

        System.out.println("Name: " + employeeName);
        System.out.println("Total High Bonuses: " + totalHighBonuses);
    }

    private static int calculateHighBonuses(int[] bonuses, int threshold) {
        int total = 0;
        for (int bonus : bonuses) {
            if (bonus > threshold) {
                total += bonus;
            }
        }
        return total;
    }
}
```
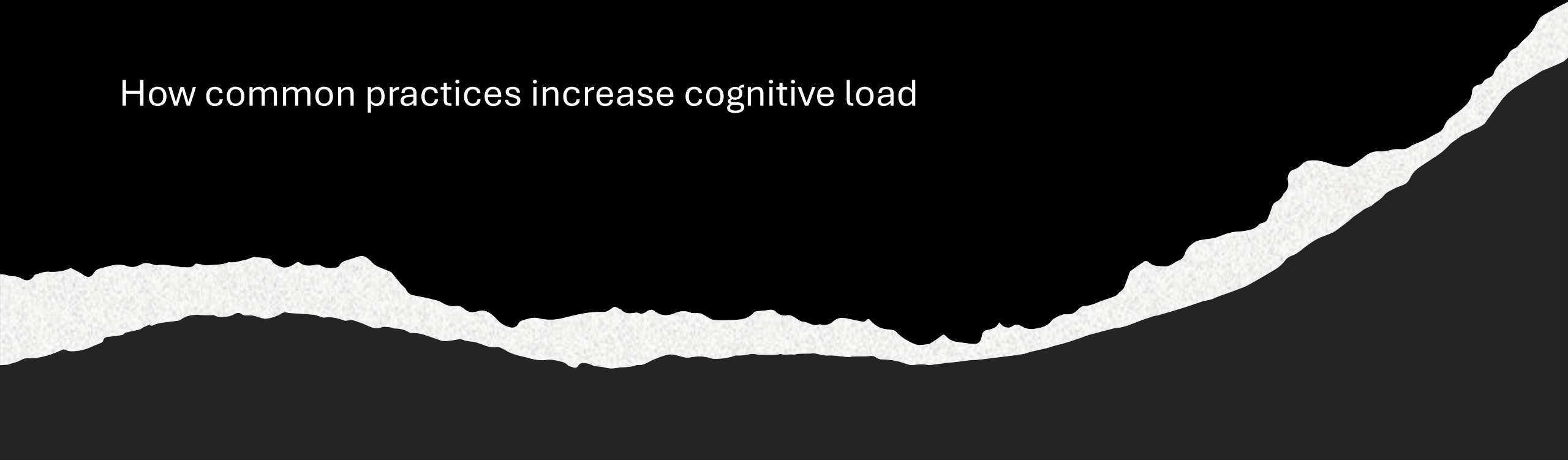
1. Descriptive naming makes the code self-explanatory.

2. Abstraction with calculateHighBonuses() separates concerns.

3. Threshold passed as parameter improves reusability and readability.

4. Comments are not even needed due to clarity from names.

Where Complexity Hides

How common practices increase cognitive load

Overuse of Abstraction Layers

Layering and abstraction are essential, but bad and unnecessary abstractions increase cognitive load with no benefit.

- **"Death by Pattern".** Applying patterns by habit rather than need (Factory class to create a single object).

- **Over-Abstraction**. Each abstraction imposes a mental cost on future readers. Linear, readable solutions might be preferable over heavily abstracted ones.

- **Too many layers, too soon.** If basic tasks require tracing through multiple wrappers, listeners, mappers, etc., the architecture might be over-engineered

Layered Service Hell
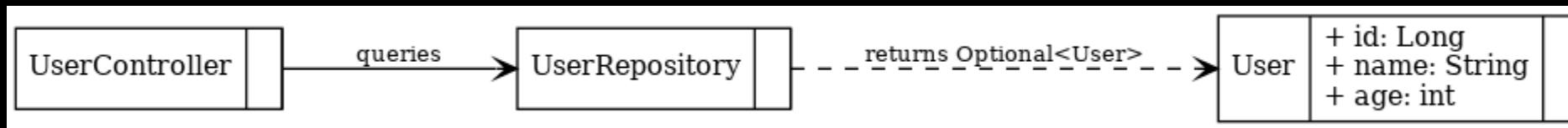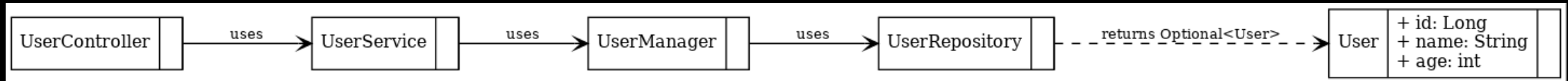
We often create unnecessary layers for no good reason
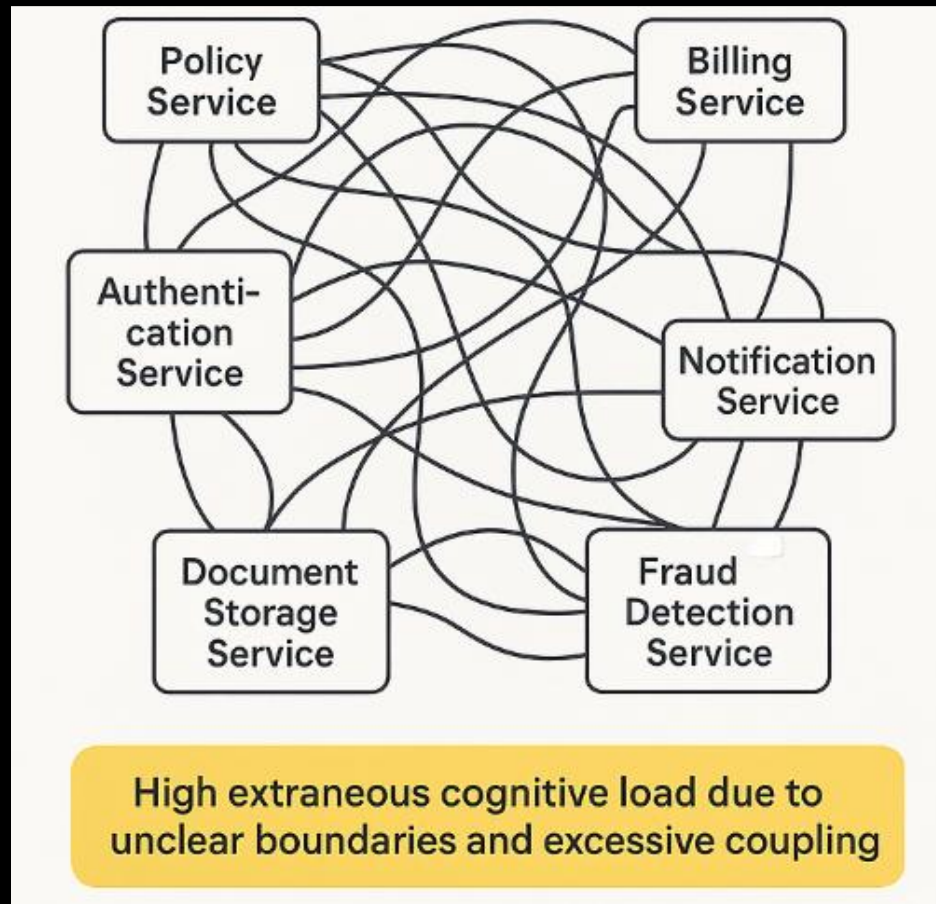
Poor Naming and Inconsistent Conventions

Poorly named variables, methods, or classes force readers to pause and decipher meaning, adding to mental effort.

Ambiguous names – "processData()"

Inconsistent Terminology – "Client" in one module, "Customer" in another!

"There are only two hard things in Computer Science: cache invalidation and naming things"

Microservices Sprawl

Policy Service
Billing Service
Authenti-cation Service
Notification Service
Document Storage Service
Fraud Detection Service

High extraneous cognitive load due to unclear boundaries and excessive coupling

Low cohesion – Services violating domain boundaries and forcing developers to think about multiple concerns at once.

Tangled mess of dependencies and interactions leading to organizational cognitive load.

Examples of Complexity Traps in Java

'Clever' Streams and Lambdas

Hard to debug, hidden NPEs, forces you to maintain a mental model of intermediate data transformations.

```
List<String> emails = users.stream()
  .filter(u -> u.isVerified())
  .map(User::getOrders)
  .flatMap(List::stream)
  .filter(o -> o.getTotal() > 100)
  .map(Order::getCustomerEmail)
  .distinct()
  .sorted()
  .collect(Collectors.toList());
```

Deeply Nested Control Structures

Every extra nesting level multiplies branching paths you must consider

```
for (User u : users) {
  if (u.isActive()) {
    while (u.hasPendingTasks()) {
      if (task.requiresApproval()) {
        for (Permission p : task.getPermissions()) {
          // ... more logic
        }
```

Complex Exception-Handling Chains

Hides failure paths, cripples diagnostics, forces guesswork.

```
try {
  doIo();
} catch (Exception e) { /* ignore */ }

catch (IOException | SQLException e) {
  throw new RuntimeException("Failed"); // drops original cause details
}
```

Reflection and Dynamic Proxies

Reflection lets you bypass normal compile-time checks, forcing you to mentally bridge the gap between code and metadata.

```
Field f = clazz.getDeclaredField("secret");
f.setAccessible(true);
Object value = f.get(obj);
```

Concurrency Constructs and Memory-Model Quirks

Data races, stale reads, sporadic failures that are impossible to reason about.

```
static List<Task> tasks = new ArrayList<>();
// updated from multiple threads

// double-checked locking without volatile:
private static Foo foo;
if (foo == null) {
  synchronized(Foo.class) {
    if (foo == null) foo = new Foo(); // broken publication
  }
}
```

"God" classes

Bloated code with no separation of concerns. Hard to test, maintain, or extend. Unpredictable ripple effects.

```
public class UserHelper {
  // do everything and anything related to users, including business logic,
  //persistence, reporting, logging, etc.
}
```

Examples of Complexity Traps in Spring

Annotation Soup & Hidden Behavior

Behavior is spread across annotations and proxies; control flow isn't visible in code

Methods stacked with @Transactional, @Retryable, @Cacheable, @Async, @CircuitBreaker, etc.

Surprises from default attributes (e.g., default rollback rules, cache keys).

Configuration/Profile Sprawl

Behavior depends on a maze of application-.yml files and env variables*

Duplicate properties across profiles; "works in dev, breaks in prod".

Mixing @Value strings all over code.

Bean Wiring Ambiguity & Field Injection

Ambiguous resolution and lifecycle surprises; testing is painful.

@Autowired on fields; multiple beans of the same type without qualifiers.

Silent selection via @Primary you forgot about.

Transaction Boundary Misuse

Lazy loading explosions, partial writes, or no rollback when expected.

@Transactional on read-only queries missing readOnly=true.

Entity graph traversal outside a transaction (LazyInitializationException).

AOP/Proxy Surprises

Proxies alter method behavior in non-obvious ways

@Transactional on private/final/static methods (won't be proxied).

Self-invocation: this.someTransactionalMethod() inside the same class does not trigger advice.

Circular Dependencies & Over-granular Services

Startup fails with BeanCurrentlyInCreationException or forces @Lazy patches.

Domain services referencing each other both ways.

"Manager/Helper/Util" layers created to break cycles.

Auto-Configuration & Starter Bloat

You accidentally pull in half the world; startup time, memory, and behavior become opaque

Adding spring-boot-starter-web when you only need scheduling.

Multiple web stacks (MVC + WebFlux) on the classpath "by accident".

Event/Listener Spaghetti

Causal chains disappear; ordering/retry semantics unclear.

Many ApplicationListener/@EventListener handlers with side effects; synchronous events doing heavy work.

Designing for Simplicity

Practical principles to reduce extraneous cognitive load in your systems

Minimize Concepts

- Reduce the number of patterns, abstractions, and layers.

- Avoid "abstraction hell" (controller → service → manager → DAO for one call).

- Every new concept is a tax on working memory.


When to Use Service, Manager and DAO

- Use fewer layers by default → Controller → Repository is enough for simple CRUD

  Controller → Repository
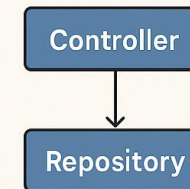
  Add a Service when you need orchestration, transactions, or cross-cutting policies (auth, retries, auditing)

  Service → Repository
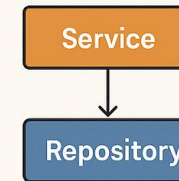
- Add a Manager (Domain Service) only if there's complex, reusable business logic that doesn't fit in one entity

  Controller → Manager → Repository
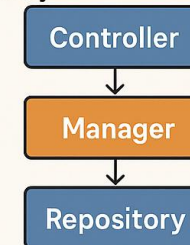
  Keep a DAO/Repository when persistence is non-trivial (joins, caching, multiple stores, legacy schema)

  Smell test: if a layer just forwards calls without adding logic, remove it

Keep Things Local

- Changes should be understandable within a small, local context.

- Avoid forcing developers to trace dependencies across 10 files or services.

- Cohesion lowers cognitive overhead.


```
outerFunction()
{
    innerFunction()
    {

}
```

- Limit the visibility of information and functionality

- Encapsulate logic within a module or component

- Define variables and functions in the smallest scope possible

Reduce Hidden Dependencies

- Make wiring and dependencies explicit (constructor injection > field injection, clear APIs > implicit magic).

- No one should have to "reverse-engineer" framework behavior.

```java
@Service
public class OrderService {

    @Autowired
    private UserService userService;    // hidden dependency

    public void placeOrder(Long userId) {
        User user = userService.findUser(userId);
        // business logic
    }
}
```

❌

```java
@Service
public class OrderService {

    private final UserService userService;

    @Autowired    // optional in recent Spring versions
    public OrderService(UserService userService) {
        this.userService = userService;    // explicit dependency
    }

    public void placeOrder(Long userId) {
        User user = userService.findUser(userId);
        // business logic
    }
}
```

✅

- The dependency is hidden — the class looks "magical."

- Harder to test (must spin up Spring or use reflection to set userService).

- No guarantee that the dependency is non-null.

- The dependency is explicit in the constructor signature.

- Easy to test (can pass a mock UserService directly).

- Guarantees immutability (final field) and makes wiring transparent.

23

Design for the Reader

- Code is read far more than it is written.

- Optimize for clarity and readability, not cleverness or terseness.

- Meaningful naming, consistent conventions, and straightforward flow.

✓
```
int daysUntilExpiration;
```

✗
```
int d;
```

✓
```
String status;
if (!flag) {
    status = "Unknown";
} else if (count > 10) {
    status = "Over";
} else {
    status = "Under";
}
```

✗
```
String status = flag ?
(count > 10 ? "Over" :
"Under") : "Unknown";
```

✓
```
List<User> activeUsers = users.stream()
    .filter(User::isActive)
    .collect(Collectors.toList());

List<User> frequentBuyers =
activeUsers.stream()
    .filter(u -> u.getOrders().size() > 5)
    .collect(Collectors.toList());

for (User user : frequentBuyers) {
    System.out.println(user.getEmail());
}
```

✗
```
users.stream()
    .filter(u ->
u.isActive() &&
u.getOrders().size() > 5)
    .map(User::getEmail)

.forEach(System.out::println
);
```

Limit Choices

- Don't provide three different ways to do the same thing.

- Pick conventions and stick to them (REST, naming standards, API patterns).

- Cuts down "decision fatigue" for developers.

Developers is left confused – are these different? Which should they use?

```
timeout: 5000
requestTimeout: 5000
connectionTimeoutMs: 5000
```

```
public static final String ACTIVE = "ACTIVE";
public static final String STATUS_ACTIVE = "ACTIVE";
enum Status { ACTIVE, INACTIVE }
```

```
System.out.println("Order placed");
logger.info("Order placed");
log.debug("Order placed");
```

```
GET /users/{id}
GET /user/{id}
GET /accounts/{id}
```

Progressive Disclosure

- Start with a simple, obvious path.

- Allow deeper complexity only when needed.

Simple path, 90% of cases. Start with this.

```
Receipt r = paymentClient.charge(customerId, amount);
```

Add depth only when real-world issues (timeouts, retries, async flows) arise.

Keeps the common path simple but makes advanced control possible.

```
Receipt r = paymentClient.charge(
    customerId,
    amount,
    ChargeOptions.builder()
        .timeout(Duration.ofSeconds(8))
        .retry(RetryPolicy.exponentialBackoff(5))
        .idempotencyKey(requestId)    // prevent double charge
        .asyncCapture(true)           // decouple from PSP latency
        .build()
);
```

Progressive disclosure hides complexity until you need it.

Fail Fast and Be Obvious

- Surface errors early → don't let nulls or hidden failures sneak downstream.

- Make contracts explicit → return Optional, throw clear exceptions, or validate inputs.

- Guide the reader → code should show clearly how to handle failure, not leave it implicit.

Catch errors early and clearly → don't let ambiguity leak into runtime surprises.

❌ Bad

```
// Can silently return null, leading to NPEs later
public User findUser(String id) {
    return userRepository.findById(id).orElse(null);
}
```

✔ Good

```
// Caller *must* handle the absence of a user
public Optional(User> findUserUserId id) {
    return OptionalUserfĩindUser();
}
userService.findUser(id)
    .ifPresentOrRelse,
        this:processUser (User not found: "):(",id)
)
```

Additional Principles for Design Simplicity

Single Responsibility (part of SOLID)

KISS, DRY, YAGNI

Bounded Contexts

Standards and Conventions

*"Simplicity is not about less code —
it's about less mental load."*

Tools and Techniques

Code analysis and documentation tools and techniques to identify and reduce extraneous complexity and cognitive load

Architecture Decision Records

ADRs are used to document and track architectural choices (including decision changes over time) in a concise, readable form.

- Capture "Why," not just "What" – ADRs record reasoning, trade-offs, and context.

- Reduce Cognitive Overhead – newcomers don't need to rediscover past decisions.

- Support Evolution – ADRs show how architecture changed and why.

- Enable Shared Understanding – teams align quickly without long history lessons.
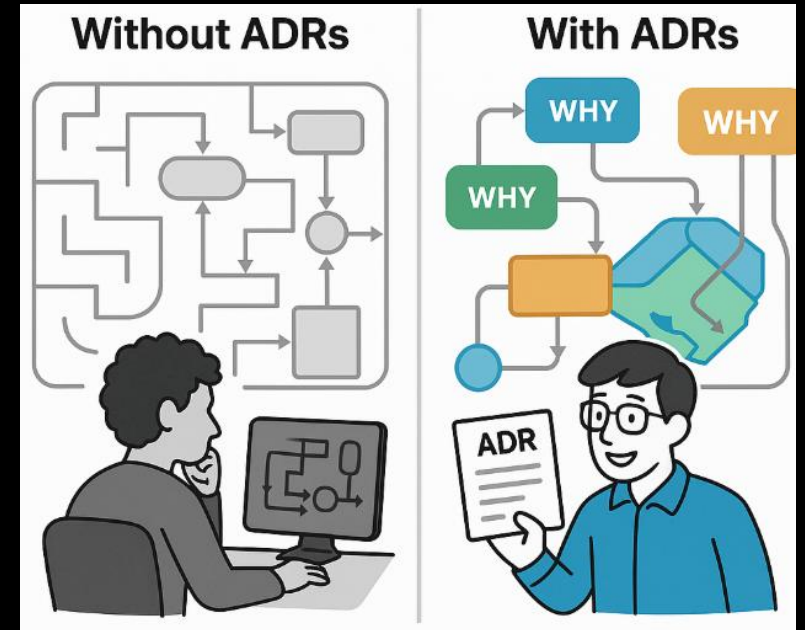
Example ADR from AWS

Diagrams as Code

Structurizr DSL documentation to diagram

- Always up to date → generated from code/config, eliminating stale diagrams.

- Single source of truth → developers don't juggle between mismatched docs and reality.

- Progressive disclosure → zoom in/out (C4 model) to avoid overwhelming detail.

- Version controlled → diagrams evolve with code, reviewable in PRs.

- Shared visual language → consistent notation reduces interpretation errors.

Static Analysis Tools in IDEs

sonarqube

PMD

Checkstyle

SpotBugs

Error Prone
(Google)
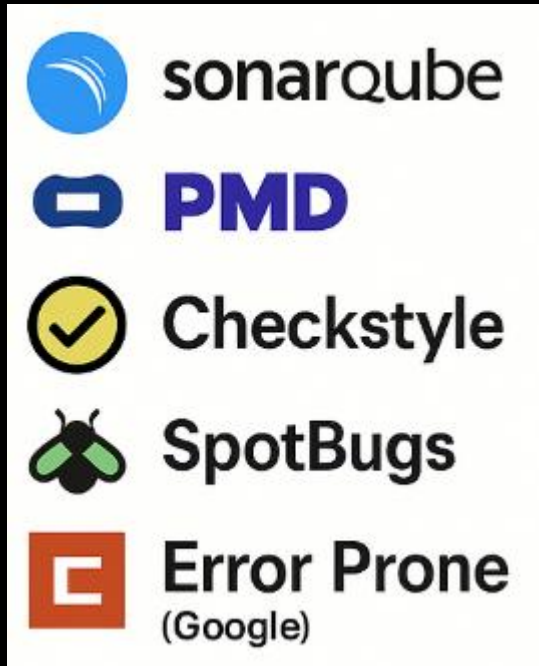
🔍 Identify code complexity issues (e.g., deep nesting, high cyclomatic complexity, long methods)

📏 Enforce coding standards & naming conventions → consistent, predictable code

🚦 Highlight potential bugs early → fail fast in the IDE before code review

🧹 Promote cleaner, more maintainable codebases → lower extraneous cognitive load

AI Assistants

GitHub Copilot

Amazon Q

CLAUDE CODE

Gemini Code Assist

ChatGPT

•**Simplify Code:** Explain complex or legacy code in plain language.

•**Spot Issues:** Flag redundancies, over-engineering, and inefficiencies.

•**Quick Answers:** Provide instant clarification without docs hunting.

•**Consistency:** Suggest clearer names and idiomatic patterns.

•**Faster Learning:** Help juniors grasp concepts with guided explanations.

Let's See Simplicity at Work

Real-World Stories
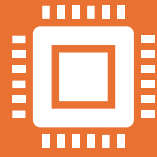
What are some examples of unnecessarily complexity from your experience?

What tools and techniques have worked for you?

What is the one design simplification principle that you feel makes the most impact?

Wrap-up and Takeaways

"Simplicity is not about less code — it's about less mental load."

Cognitive Load Matters
Code is read more than it's written:
unnecessary complexity drains mental energy

Complexity Hides Everywhere
Layers, poor naming, "God" classes,
annotation soup, microservices sprawl

Design for the Reader
Favor clarity, consistency, and explicitness
over cleverness or hidden magic

Simplicity Wins
Minimize concepts, reduce hidden dependencies,
fail fast with obvious signals

Tools Help
ADRs, diagrams-as-code, static analysis,
AI assistants reduce overhead

Resources

The Cognitive Load Theory in Software Development

"Why our brains struggle with complex code—and how Cognitive Load Theory can guide us to write software that's easier to learn, understand, and maintain."

Navigating Cognitive Load in Software Development: Experiences, Strategies, and Future Tools for Enhanced Mental Workflow

"Inside a developer's mind: how we juggle complexity, manage mental strain, and what tools can lighten the load

Cognitive Load is what matters

This GitHub repo by Zakir Ullin is widely recognized as one of the most practical resources for understanding and applying cognitive load concepts in real-world coding.