FOUNDATIONS

Why Structure Is Everything

William Christopher Anderson

Foundations

Why Structure Is Everything

William Christopher Anderson

Copyright © 2026 William Christopher Anderson

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author.

First Edition

Printed in the United States of America

Dedication

For Fox, Benjamin, and Chloe —
who made everything I built worth building,
and who will one day understand that the structure a person creates
is never really about the code.

And for Sarah.

You have been present for the years that are in this book
and the years that are not.
You have seen the work and the cost of it.
You have held things together when I was not capable of holding them myself,

and you have done it without asking me to be grateful,
which made me more grateful than I knew how to say.

The chapters in here about family and faith and what survives
—
they exist because of what you gave.
More than any reasonable person should have had to give.

I do not know what shape our life takes after this.
What I know is that you are the love of my life.
That was true before I understood it.
It will be true in whatever future either of us inhabits.
If I lose you, it will be my life's great loss —
not because of what we built together,
but because of what you are.

This book is yours.
It was always yours.
Whatever comes.

Table of Contents

Preface 8

PART I: BEFORE THE FRAMEWORK

Chapter 1: Thirty Moves 13

Chapter 2: December 2nd 34

Chapter 3: The Classifieds 52

PART II: PATTERN LANGUAGE

Chapter 4: One Structure, Every Layer COMING SOON

Chapter 5: The First Decomposition COMING SOON

Chapter 6: When the Music Stops COMING SOON

PART III: THE FOUR PILLARS

Chapter 7: What Changes COMING SOON

Chapter 8: Through Their Eyes COMING SOON

Chapter 9: Where Things Break COMING SOON

Chapter 10: The Weight of a Project COMING SOON

Chapter 11: Four Lenses, One View COMING SOON

Chapter 12: The Harmony Emerges COMING SOON

PART IV: PRESSURE TESTING

Chapter 13: The System That Fought Back COMING SOON

Chapter 14: Breaking and Rebuilding COMING SOON
Chapter 15: What Survived COMING SOON

PART V: EXTENSIONS

Chapter 16: Compiled Context COMING SOON
Chapter 17: Swarm Intelligence COMING SOON
Chapter 18: The Complete Picture COMING SOON

PART VI: APPLICATION

Chapter 19: At Scale COMING SOON
Chapter 20: Teaching the Framework COMING SOON
Chapter 21: The Living System COMING SOON

PART VII: THE HUMAN ARCHITECTURE

Chapter 22: The Architecture of a Family COMING SOON
Chapter 23: Faith and Structure COMING SOON

PART VIII: FOUNDATIONS

Chapter 24: Why Structure Is Everything COMING SOON
Epilogue COMING SOON
About the Author COMING SOON

APPENDICES

Appendix A: Harmonic Design 71
Appendix B: Volatility-Based Decomposition 168
Appendix C: Experience-Based Decomposition 225
Appendix D: Boundary-Driven Testing 284
Appendix E: Project Design 339
Appendix F: Compiled Context Runtime 391
Appendix G: Swarm Architecture 483

Preface: What This Book Is

> *The test of a first-rate intelligence is the ability to hold two opposed ideas in the mind at the same time, and still retain the ability to function.*
>
> — F. Scott Fitzgerald, *The Crack-Up*

There is a question at the center of software architecture that I spent twenty years learning to ask precisely: how do you build something that survives change?

The naive answer is build it right the first time. That answer is wrong. Systems do not fail because they were designed badly at the start. They fail because change accumulates faster than the architecture can absorb it. The requirements shift, the

business changes direction, the team turns over, the technology evolves — and the structure that was perfectly adequate for what you knew when you built it is suddenly fighting everything you learned after. The question is not how to design correctly. It is how to design in a way that can survive being wrong.

I have spent the same twenty years applying that question to my own life without knowing I was doing it.

This book is two things simultaneously. It is a rigorous software engineering framework — Harmonic Design — built from twenty years of practice across some of the largest systems in commercial technology. And it is the biography of the life that produced it: a childhood of roughly thirty moves between my parents' separate collapses, a library that was always open when everything else was not, text-based worlds I built on a public terminal before I understood what programming was, a son I was raising at twenty-one with no instruction manual, and a marriage I almost lost and may yet — a framework finished in a motel room in April 2026 while all of that was still in motion.

Neither piece is decorative. The biography is not a warm-up for the framework. The framework is not the payoff for the biography. They are two readings of the same thing.

The organizing principle of the framework — design around what changes, not around what currently is — turns out to be the organizing principle of a life too. That is not a metaphor. It is the argument.

This is not a memoir that uses software as a clever device. The technical content is real and rigorous; the whitepapers behind it have been applied at scale, and engineers who read it should find it useful. It is also not a framework book that uses biography as color. The biographical content is honest, including the parts that are not easy to tell, because the framework cannot be fully understood without the conditions that produced the questions it answers.

Some of what you will find here is ugly. Addiction. Violence. A single incident of abuse at fourteen that my mother failed to address, and that I am still measuring the cost of. A manic episode in which I gave away $75,000 and threatened to kill myself. A marriage being rebuilt in real time as I write this, with the outcome not yet settled. None of it is included for shock. All of it is load-bearing. Remove it and the framework floats — technically coherent but unanchored, interesting on the page and inert in practice.

The book is organized in eight parts. Each contains both the biographical arc and the framework element it surfaces. I have not arranged the biography to flatter the framework, or shaped the framework to resolve

the biography cleanly. The seams show in places. I think that is correct. Clean seams — in architecture and in life — often mean you have concealed rather than resolved.

One more thing before the story starts.

I grew up knowing that structure was not guaranteed. My father taught me to ask good questions; he also attempted suicide with a butcher knife in our kitchen when I was six. My mother loved me; she also failed me in the specific and concrete way that a parent can fail a child. The library was always open, and it was sometimes the only stable thing I had. I left home at fifteen. I have been building structure — in code, in organizations, in rooms where people are trying to make something that lasts — ever since.

The framework in this book is called Harmonic Design. A vibrating string produces not one frequency but many — a fundamental tone and a series of harmonics above it that do not contradict the fundamental but arise from the same underlying physics. That is what I set out to build: an approach to software where the architecture, the experience design, the testing strategy, and the project plan all arise from the same underlying principle, each one reinforcing the others, nothing fighting anything else.

I wanted the same thing for my life. I am still working on it.

This is where I am. This is what I know. Here is what it cost to find it out.

Part I: Before the Framework

Chapter 1: Thirty Moves

1985–2003

✦ ✦ ✦

Home is where one starts from.

— T.S. Eliot, “East Coker”

Every day when I was four or five years old, in the yard of our house in Cedarville, I let a turtle go.

I’d carry it around for a while — small, muddy, already pulling its head in — and then set it loose near the edge of the grass. That evening, when Dad came home from work, he’d hold out a turtle for me to take. A new one, I thought. I’d take it like it was treasure.

I didn't find out until years later that it was the same turtle. He was catching it every night after I let it go, bringing it back when he came home because he knew how much I loved it. He kept the ruse going for months — maybe longer. The patience of it, the consistency, the care about what it cost me not to notice — that was my father before everything else. The man who thought through the details of small kindnesses and executed them without being found out.

He'd let me sleep between him and Mom when we had no money for a room of my own. At bedtime, he'd answer every question I had — unlimited, for as long as I could think of them. Then, when he finally got tired, he'd tell me I had twenty questions left, and I'd lie there in the dark thinking carefully about which twenty were worth spending.

Sometimes, while I was thinking, I'd just fall asleep — as happy as I've ever been.

I believed he was the smartest, coolest man alive. I wasn't entirely wrong. I just didn't know yet what he was also capable of.

✦ ✦ ✦

I was born July 5th, 1985, in Paris, Arkansas — a small town in Logan County that most people have never heard of and that I left before I was old enough to remember it clearly. My earliest real memories are

from a few years later: a yellow learning game shaped like an elephant, the trunk curling along its edge, that taught me letters and numbers. I was reading by three or four. I'd read the backs of cereal boxes out loud to my sisters and they'd cheer, and I'd do it again. That dynamic — my capability stepping into someone else's need — became a pattern that ran through everything afterward, and was, in time, a burden as much as a gift.

There were five of us. Nikki was four years older. Jessie was three years older. I was the only boy. Savanna would come years later, half-sister, born to a different father during the wreckage of what followed. But in the early years it was the five of us: Mom and Dad, two daughters, one son, trying to hold a household together with insufficient money and insufficient tools and a growing problem with alcohol and drugs that neither of them was equipped to address.

Before this story goes where it goes: I love them — all of them — deeply, in ways I have never fully known how to say. I did not and do not have the relationships I wanted with any of them. Both of those things live in me at the same time, and they do not cancel each other out. The love was not a consolation prize for the missing relationship. It was its own real thing, and it is still there.

We passed through Cedarville briefly — long enough for me to remember a screened-in porch where you could sit and listen to rain without getting wet — but Van Buren was where we landed, where I grew up, where everything that matters in this chapter happened. Fort Smith, just across the river, was where I eventually found the pool hall and something like independence.

But before all of it: Mom singing to us — not well, but without a trace of self-consciousness about it — taking us places, present in a way that children file away without knowing they're filing it.

I'm telling you this now, before the rest of it, because what follows tends to crowd out what came before. Don't let it. The porch and the singing and the warmth — those happened. The collapse happened too. Both things were true about my family, and the difficulty of holding them together at the same time is something I have spent forty years working out.

✦ ✦ ✦

By age five we were living at Flat Rock Court, a trailer park in Van Buren. Mom and Dad were both drinking hard by then, and Dad was already experienced with meth — recreational at that point, but experienced. They smoked prodigious amounts of pot too, though I understand well enough how marijuana can take the edge off a hard situation, and I'll leave it at that.

The fights were getting worse — two people under that kind of pressure with nowhere to go, grinding each other down.

One night they got into it and Dad smashed Mom's face through a glass coffee table. I heard the sound of it shattering. She staggered to her knees, wobbly, and I saw her face — I still remember her tooth jutting out of her bottom lip, the blood, the way she looked like something had gone wrong with the world at a structural level. Then Aunt Amy came through the door screaming: *"You killed my sister, you son of a bitch."* That's when I ran. I hid under a neighbor's porch and stayed there for hours, not knowing what to do, not yet capable of understanding what I was waiting for.

He hadn't killed her. But he had killed something between them.

After that we moved. The new place was an actual house — green, with a front and back yard so overgrown the bushes reached higher than the windows. Dad and Melvin spent a week trimming them down together, clearing the yard, getting it livable. It was halfway condemned, probably, but it wasn't on cinder blocks, and after Flat Rock Court that felt like something. We moved in and it felt like a fresh start.

We called it the hell house for the rest of our lives. But moving in, it didn't feel like that yet.

That came later. Melvin stayed. He was Dad's friend — which is the detail that makes everything else worse. He came from a family where the father made the boys fight each other for sport. Most of them ended up convicted of something. Most of them ended up addicted to something. He and Mom started an affair under the same roof where Dad was still trying to hold a marriage together, with his friend sleeping down the hall.

The fights were no longer about pressure. They were about the reason for the pressure standing right there in the room.

✦ ✦ ✦

The divorce happened when I was in first grade — JJ Izzard Elementary, Van Buren. I came home to find Nikki and Jessie sitting on the porch swing at the hell house, crying. I walked over and sat down and started crying too, and I didn't yet know why. Inside, Mom and Dad were screaming. Aunt Amy was there, also screaming. At some point Mom and Amy rushed out with my sisters — but I stayed with Dad.

I watched him fall apart. Then he pulled a butcher knife from the block and began carving into his chest, saying that Mom was his heart and he was going to finish what her leaving had started. I was terrified in the specific bodily way that a six-year-old is terrified — it was in my hands and my stomach, not just my

understanding. I told him: if you kill yourself, I'm going to kill myself too. He said that if I killed myself, he'd come back and kill me. The logic was recursive and made no sense. It worked anyway.

I stopped — and, more importantly, so did he.

We sat there and waited for Grandma Lucas to come from Kansas and take him. While we waited I cried with him and sometimes went out to the tire swing he'd hung in the front yard. When Grandma Lucas arrived, Mom had returned, and I had to choose which parent to go with. I started toward Dad. He told me to stay with my mother and be the man of the house.

I took that seriously, and I have been trying to be that man ever since.

That same year — just after Christmas, still six years old — Mom and Dad attempted a reconciliation. They moved into Jesse Turner Terrace, low-income housing just up the hill from the hell house, and for a while, I don't know how long, things felt like they might hold. Time is unreliable in childhood, especially in childhood like mine. All five of us in the same space, the fighting stopped, the house quiet in the evenings. That is my last clear memory of genuine family happiness. I used to think that period must have been

longer than it actually was. People tend to expand the good periods in memory and compress the bad ones. I think I do the opposite.

Then Mom discovered she was pregnant with Melvin's child.

She left again — Nikki chased her down the road begging, and she went anyway — and she would stay married to Melvin for over twenty years.

The day after — or maybe a few days later, the timing is uncertain — I came home from school, King Elementary by then, expecting to watch cartoons with Dad the way we always did: *Animaniacs*, *Tiny Toons*, *Pinky and the Brain*, the whole afternoon block. He was asleep and wouldn't wake up. I called 911. They took him away and pumped his stomach. He had overdosed intentionally.

That was not the last time. When I was eight or nine he took a Ginsu knife to his wrist after a fight with his second wife, Anne, and nearly bled out. I wasn't there — I only got to visit him in the hospital after. I remember sitting with him and asking: *"Daddy, why do you hurt yourself?"* Having gazed long enough at my own shortcomings, I understand that why all too well. The surgeries that followed restored most — but not all — of the function in his hand.

✦ ✦ ✦

Before the divorce, my parents moved plenty — I can count about six distinct houses, trailers, and apartments across those early years. Still too frequent for a child to feel settled, but nothing that would prepare you for what came next.

The real chaos started when Mom and Melvin made it permanent — that's when the thirty moves happened. Not because my parents couldn't decide where to put me — because we were always being put out. Money would run out. A landlord would reach the end of his patience. A neighbor would file a complaint, or the wrong person would get into the wrong argument with the wrong person, or Mom or Melvin would just get itchy feet and decide that somewhere else had to be better than here. There was always a reason. Sometimes I wouldn't even bother pulling things out of boxes — just leave them in there, because boxes were easier to move when the time came, and the time always came.

Between ages six and fifteen I moved over thirty times. Not between two houses — to thirty different addresses. Houses, trailers, and apartments. Whoever would take us.

New landlords, new neighborhoods, new school districts, new kids who didn't know me and old kids I had to leave behind. No bedroom that was mine for more than a few months. No wall I could put a nail in without wondering if I'd have to patch it before we

left. Sometimes we stayed with a relative for a stretch. Sometimes it was just wherever someone would take us.

We never stopped long enough for any of it to feel permanent — which meant we never stopped long enough for me to believe that permanent was a thing that existed.

What held was the library. In the early years it was the school library — whatever school I happened to be in, there was always a library, and it was always the same kind of place. Around twelve, I found the public library, and that was something else entirely. Free. Always open. No questions asked, no adult required to bring you or come back for you. A computer available every day. You could walk in with nothing and leave knowing something.

That was a kind of structural guarantee I hadn't found anywhere else, and I understood it without being able to say what I understood.

I read everything I could reach. At home — whatever home it currently was — we had a few volumes of Encyclopaedia Britannica. The "A" volume introduced me to Atlantis and I spent a year obsessed with lost civilizations. Greek mythology. Huckleberry Finn. The Bible. Whatever was on the shelf or the floor or the windowsill. Those nights with Dad had done their work: I thought of reading as thinking, and I thought of thinking as the thing that differentiated

people who could handle whatever came from people who couldn't. I was building something out of that belief. I just didn't have a name for it yet.

✦ ✦ ✦

Around second grade we were in Kibler — a small rural town between Van Buren and Alma. A horse kicked me in the head in a field near the trailer. A neighbor watching from his kitchen window saw a small body arc through the air and brought me to Mom, telling her the boy might be dead. My only real memory from my entire second-grade year is waking up on the porch, spitting out teeth, and saying: *"Mama, that damn horse kicked out my teeth"* — and then going dark.

I had nearly overcome a speech impediment through speech therapy in kindergarten and first grade. The kick regressed it completely. I carried it until I was twenty-five, when I could finally afford my own insurance and private therapy. Every school move reset the clock on accessing services — by the time a new district's therapist found me in the roster, we'd moved again.

Also in Kibler: I ate part of an elephant ear plant. The calcium oxalate crystals caused my tongue to swell badly enough to partially obstruct my airway.

Both of these incidents were survivable. Both pointed to the same condition: a child who was not being watched.

Mom was still present during this period, but the woman I remembered from Cedarville was beginning her long drift. Meth and pills were pulling her incrementally further. She would find her way back — but not until I was in my thirties, and that story belongs later in this book.

✦ ✦ ✦

The best two years of my childhood were the years I got to live with my childhood best friend Josh.

Josh Broadway was my best friend first — that's the part that mattered. His mother was Debbie Newman, and it was through me and Josh that Dad eventually met her. Josh was kind and steady, a little bit rigid — the kind I've come to recognize, looking back and knowing him now, as neurodivergent. He'd tell you himself.

I loved Debbie like a second mother. She took me in without ceremony, fed me, kept the lights on, and never once made me feel like a problem she hadn't planned for. But she wasn't soft about it either. She knew I was bright, she told me so, and then she told me that bright didn't mean anything if I wasn't willing

to work for it. She held me accountable in a way almost no adult in my life had. That meant something. It still does.

It was the most stable stretch I'd had: one address for two consecutive years, reliably there in the mornings. Not without chaos — nowhere in my life was without chaos — but the boxes stayed put, and after everything that had come before, that was its own kind of luxury.

I was drinking by twelve. Nothing catastrophic yet, nothing I couldn't account for. Dumb pranks, running with friends, once tipping over a buffalo sculpture at school. The energy of a kid with too much intelligence and nowhere to put it, who had learned early that the only reliable way to get what he needed was to take it. I was mostly happy during those years. I want to say that plainly, because mostly happy was not a thing I could say about most of my childhood.

Also during those years: the computer lab at school. *Oregon Trail* was there, but the program that mattered wasn't *Oregon Trail*. It was a conversational one — you could type to it, and it would respond. Insult it and it would tell you to go fly a kite. Be kind to it and it would be kind back. A machine with personality. I couldn't stop thinking about it. It made visible something I had been trying to articulate without success — a world that responded to what you put into it. An intelligence — simulated, artificial,

but consistent and responsive — that you could build a relationship with. I thought about it long after I walked out of that computer lab.

✦ ✦ ✦

At twelve and thirteen I found MUDs — Multi-User Dungeons — text-based online worlds accessible through Telnet at the library. Legends of Merlin. Pandora's Box. My favorite: Rapture of Oblivion, which I still think had the best builder community of any of them.

At fourteen, the admins of one of those servers invited me to build. I lied about my age to get in — easier to do in those days. The tools were online: you'd create a room, write its description, populate it with mobs, script their behavior, set triggers and flags. A cursed item needed its flag. A weapon needed its damage range. A mob needed to know what it said when it aggro'd and what it dropped when it died. I built obsessively. They saw what I was producing and invited me to go further — to actually code.

That meant SSH, a Putty client, compiling changes, submitting for review. The server owner would accept or reject and tell me why — it was the first time anyone had evaluated my work against a standard and explained the gap, and I taught myself all of it because the game was on the other side.

That is where software began for me. Not in a classroom. Not from a mentor with a curriculum. From a text-based world that needed building, and an invitation to come work on it, and the recognition that the tools for getting there were learnable if you wanted them badly enough. I wanted them badly enough.

Dad exploded the situation with Debbie that same year — said something unjustifiable, did something he couldn't walk back, left in the pattern he always followed. I went back to Mom's.

✦ ✦ ✦

At fourteen, my uncle James — Mom's brother — molested me. Single incident. My immediate physical response was to go to the bathroom and vomit. That response — my body's particular way of handling what it cannot metabolize — became a physiological pattern I carried for most of my adult life.

I told Mom, and nothing was done.

This is the real reason I left home at fifteen. Not the general chaos, not the poverty, not the accumulated instability of all of it. The direct cause was being violated and then being failed by the one person who should have protected me. I have not always told it in that order, and I have not always been

willing to state the cause that plainly. I am stating it now because this book asks for the parts that are load-bearing, and this is one of them.

✦ ✦ ✦

Sophomore year — Van Buren High School, if you want the official record — I left the house. I did not stop attending school. I crashed on couches: Logan Satterfield, Travis Schmitt, Josh Broadway, whoever would have me for more than a few nights. I slept in my car when it was necessary. When the weather was bad and no couch was available I would sometimes go back to Mom's, but the house was not home and I was clear in my own mind about the difference.

My financial survival ran through The Family Cue Center, a pool hall in Fort Smith owned by a man named Tony. Tony let me brush tables during busy hours in exchange for free table time and paid me twenty dollars a night to clean up at closing. Twenty dollars was gas and food. I played constantly and got good fast — nothing but time, and something worth putting it into. I fell in love with the game the way I had fallen in love with the MUDs and the library and anything else that rewarded attention and thought and practice with proportional returns.

I still play, and I still play well.

✦ ✦ ✦

In sophomore English I had a teacher named Mr. Rotert. White hair, short, a little soft in the middle, a deep rich voice that commanded the room when he read. He was kind the way a few teachers are — not generically pleasant, but actually paying attention. He watched me memorize the Marc Antony funeral speech from *Julius Caesar* in a single class period and suggested I enroll in honors and AP.

I did — and my junior year I got straight As in every class.

This was not surprising to me — I had always understood that I could do academic work when I was present enough to do it. What was surprising was that someone looked directly at me and said: you should be in the harder thing. You belong there. Mr. Rotert was not the first person to see me, but he was the first person with institutional standing to act on it, and there is a difference. It mattered more than he knew. It probably mattered more than he ever realized.

Mrs. Davis taught junior and senior English — full of snark and sarcasm, devilishly smart, refusing to protect you from ideas and expecting you to do the same. She died young, fifteen years or so after I graduated, and I think about her more than she would have expected. She eclipsed even Rotert.

I had art class with a teacher named Mr. Lowe. We did not get along. Once I secured the credit I needed for graduation I stopped attending. I'd stay out

shooting pool until two in the morning, sleep, arrive at school by nine for second period. Van Buren eventually kicked me out of the honors and AP program for the absences. I graduated anyway in May 2003, which I considered a reasonable outcome given the circumstances.

✦ ✦ ✦

At fifteen — the same year I left home — I walked into a church and found something I had not known I was looking for.

February 18th, 2001. Some dates you just keep. Whatever the correct theological framing of what happened, what I felt was this: for the first time in my life, I knew what it was to be loved fiercely and unconditionally. My relationship with faith has gotten complicated in the years since, and I'm not going to let that complexity reach backward and hollow out something that was full when it happened.

I became Vice President of Partners in Christ at school. The President was a girl named Sarah Rose Bishop, whom I had known since second grade.

That is a detail I will return to.

Dad was in a meth bender through most of my junior year and returned to Kansas — absent in body, and when present, absent in spirit. Faith filled the structural hole. It gave me a community, a vocabulary for what I was experiencing, a framework for

understanding the chaos I had grown up inside. I went in all the way. I always did, with anything that offered the possibility of comprehension.

Senior year the questions arrived. I found it easier to empathize with Satan than with God — Satan understood failure from the inside, constitutively, whereas an omnipotent perfect God had no frame of reference for it. I said this in Christian community. The community spread rumors that I was worshiping the devil. It was my first round of theological deconstruction, and it was not my last, and I look back on the questions themselves with considerably more respect than I look back on the community's response to them.

✦ ✦ ✦

I graduated in May 2003 knowing three things about myself that I had demonstrated in enough contexts to trust as real and not situational. I could memorize anything and retrieve it on demand — speeches, tests, the encyclopedia volumes I'd worked through in the library, the Marc Antony speech memorized in a single class period. I could build worlds in text: rooms and characters and spells for game servers that real people played in, using tools I had taught myself on machines that didn't belong to me, producing work that was used and valued. And I knew — not in language but in the body — that nothing holds together on its own,

that the things worth having have to be built and maintained and defended, that nothing stays without someone deciding it will stay. I had watched enough things fall apart to understand that the falling apart was never an accident. It was always the absence of a decision.

There is a principle at the center of everything I will argue in this book: stability is not the default condition of things. It is designed, or it doesn't exist.

Moving thirty times before you're fifteen teaches you that before it teaches you anything else. Not the lesson itself, not the vocabulary for it, but the instinct underneath it — a watchfulness that develops when nothing around you stays in place. You learn to look for what actually holds versus what merely looks structural. You learn the difference between a wall that is load-bearing and a wall that is decoration, because at some point you have leaned on the wrong one.

The library held. Books held. The pool table held. A screen full of text in a game world nobody could see you playing — that held too, because it responded to what you put into it and it was always there when you came back.

He was my father and I loved him. The knife in the kitchen was also real. I am not telling you that the chaos produced the framework — that would be too

clean, too much like a story someone invented to make sense of a life in retrospect. What I am telling you is that the question the framework answers — *how do you build something that survives change?* — was not a question I arrived at from a textbook. I arrived at it from the inside. I had been living inside it since I was six years old and watching my father decide not to die on the kitchen floor.

The rest of this book is the answer I eventually built.

Chapter 2: December 2nd

2003–2006

✦ ✦ ✦

> *Life can only be understood backwards; but it must be lived forwards.*
>
> — Søren Kierkegaard

The card declines on a Monday night.

I'm standing in the dining hall at Arkansas Tech University, holding a tray I haven't paid for yet, and the machine says no — not a thirty-dollar shortfall that clears after payday, but empty, something reached in and scooped it clean. I know what happened before I

put the card back in my wallet: my roommate Alex had borrowed against it — the meal card, drained by the end of the semester — and I'd been drunk most of that semester and couldn't give you an accurate accounting of when or how much, and neither of us had been watching the number. What gets me standing there in the fluorescent light isn't the money, exactly — the money is almost beside the point — it's the particular quality of the moment, the realization that the structure I had assembled around myself had a hole in it I hadn't inspected. I had worked to get into this place and done the academic work when I was present enough to do it, but I hadn't been present enough, not in the ways a university requires — not on academic probation yet, but I could see it from where I was standing, the gap between good test scores and poor attendance, and I knew what that trajectory looked like.

December 1st, 2003. I am eighteen years old.

The next morning — December 2nd — my friend Chris Angel finds me and tells me the drop window is still open — if I go to the registrar today, I can preserve the transcript, keep the record clean enough to come back later. He offers to walk me there himself.

I go back to the room and pack everything I own into the Ford Escort hatchback in the lot. I have never owned enough to make packing slow. I am going to New York. I am going to write a novel called *Winds of Change*.

I drive back to Dyer, and on the way I get pulled over for no insurance, which is how New York stops being where I am going. I leave approximately forty library books checked out at Arkansas Tech, along with whatever fines had accumulated — I am not entirely certain I ever paid those — and I would like to tell you I had a plan beyond the novel, but I did not.

✦ ✦ ✦

The novel does not get written, and what I have instead is a bed at Bubba and Gina's — a trailer in Dyer, five miles from nothing. Richard Dale Harris Jr. is my cousin; Gina is his wife. Neither of them is more than a few years older than me, and Gina has been close with my sisters since before I was paying attention. I had lived with them the summer before ATU, and the warmth in that trailer is genuine and unsurprised. I eat everything in reach. I always have — hollow legs, my mother used to say, and she was right about that if not much else at the time. There is real love in that trailer. For a while we are a sweet,

unconventional little family, the three of us — and I did not know then how much I needed that, or how rarely I had had it.

Some of that went back further than the trailer. Bubba had been a fixed point in the story I told about myself long before I showed up on his doorstep with nothing. We rode the same bus out of Flat Rock when I was small, the older kids at the back establishing order the way only cruel and shortsighted children can. In my telling, he was the kind of person who knew exactly where he stood and made that standing visible — someone who moved through the world with an ease I could not locate in myself, who seemed to belong wherever he was without working for it. I remember a kid picking on me and Bubba punching him in the nose. I filed it away as proof of the story I was already telling, and the mythology around him held for twenty years.

The real version is smaller: he and his friends teased me too, picked on me the way older kids tend to do with the one who follows them around, not cruel but not gentle either. I followed them anyway. Not the figure from the bus. Just a kid, probably carrying his own version of whatever I was carrying and better at not showing it — the coolest person I had ever seen, a sort of hero to the little boy I was. In those years he was still himself, and that was enough. Addiction found him four or five years after this. He broke his

ankle. A doctor prescribed Lorcets, which is what doctors did then; the dangers of opiates were not yet the understood thing they would become. Circumstance and genetic predisposition did the rest. He did not want to be an addict. He did not choose it. That is the part I think about — not the ending, but how ordinary the beginning was, how little agency any of it required from him. It did not consume him all at once. That is not what addiction does. It eats away at you from the inside, a kind of spiritual and psychological cancer — slow enough that you can almost not notice it until you cannot miss it anymore. Almost twenty years on, his family would collapse. Last I heard, he was somewhere in Texas.

Dyer in winter is exactly what it sounds like — nothing open, nothing happening, the highway running through without stopping. There is a truck stop diner on that highway, open twenty-four hours, and I pick up night shifts when they are short a body. You learn the regular customers fast: which truckers want coffee kept full without being asked, which ones want to talk at two in the morning and which ones don't, which ones tip and which ones are going to leave you three percent and feel righteous about it. The work is not complicated but it has a cadence, and I go home smelling like grease and coffee with enough to eat and not much more.

When I am not working I am on the Xbox — *Knights of the Old Republic*, which I play for hours at a stretch because the world it builds is coherent and mine is not — or at Midland Bowl for karaoke, where I go up exactly twice after enough ill-gotten beer has quieted the part of me that monitors every syllable before it leaves my mouth. The room will hold you if you go up like you mean it, and I find that to be true of more situations than karaoke.

Mostly I read: the Van Buren public library, twenty minutes up 64 Highway, whatever is in the trailer, Bubba's comic books — Superman, mostly, which is how I first encountered my favorite fictional character. Not because of what he can do. Because of what he chooses not to do. A being with the power to end anything, who decides instead to hold the line. What I am actually reading for is the sensation of being inside a world with rules that hold, a place where if you understand enough of the logic, events stop being arbitrary. I have been trying to find that sensation in the actual world for most of my life, and the books are faster.

✦ ✦ ✦

In the summer of 2004, Sarah Bishop comes back into my life — the same Sarah who was President of Partners in Christ at Van Buren High while I was Vice President, the same girl I had known since second

grade. The year before, her parents had divorced: her father Don was a Southern Baptist preacher, her pastor her entire life, the man who stood at the front of a church every Sunday and talked about grace and permanence, and now he was in Michigan. She was at her mom's new place, still sorting out what that meant.

We spent two months getting to know each other properly — neither of us in a hurry, both with more to say than we usually had occasion to say. The poetry was serious — our own work, not assigned, not performed for an audience, just things we wrote and handed over and the other person read, which is its own kind of trust. We talked about *The Lord of the Rings* like it mattered, which it did — not the movies, the books, the actual architecture of the world Tolkien built and what it meant that he had built it with such completeness. We put Counting Crows on and didn't say anything at all. We were both learning guitar, which is where the similarity ended: she had rhythm and could carry a tune, things I have not been blessed with, and I teased her about being better at it and she teased me right back.

She is serious and thoughtful in a way that does not announce itself, and I found myself in those months doing something I had rarely done with another person — not performing, not managing the impression I was making, actually thinking about

things and then saying what I actually thought and waiting to see what she said back. That is a rarer experience than it sounds. I was nineteen and she was the first person I had met who made it feel normal.

I came close to falling in love with her, and the only thing missing was reciprocity. She still doesn't agree we dated, and I have made peace with this — you carry the standard someone like that sets, and I still do.

Early 2005, the phone rings and it is Grandma Lucas.

Grandma Lucas is my dad's mother. Before I was ten she was just the grandmother in Kansas — the one who took us to Worlds of Fun and bought things and kept a comfortable distance. She wasn't rich; she just had more than we did, which at the time felt like the same thing. On my mom's side was Grandma Harris — Mary Ruth — who never took much interest in us kids, and Grandma Lucas was different from that, but the difference took time. It started when I was ten, and I had spent three or four summers with her between ages ten and fourteen — summers that felt categorically different from everything else in my life. A house in Kansas where the lights stayed on and dinner appeared at the same time every night. A woman with a master's degree in psychiatric nursing and two or three packs of Virginia Slims per day who

sat across from me and asked real questions and listened to real answers. The best smoker's laugh I have ever heard — a laugh that costs something and is worth it. She had bought me my first Nikes from a mail-order catalog when I was ten — you circled what you wanted and it arrived at the door six to eight weeks later — and I felt like hot shit. My mother had reported $9,800 in income for a family of five that year.

What she was doing was seeing me — the brightness and the sadness both, held in the same gaze and loved for what they were. Not the capable public face I had been constructing since I was six, but the whole thing. She loved big — a fixed point, a coordinate established, a distance from which everything else could be measured.

She says: come to Kansas. She will help me get back into college — financial aid, community college, a place to stay while I figure out the next step. She has been doing this since I was ten, finding the next step for me, clearing enough ground that I could see it. More plan than I have had in over a year. More plan, honestly, than I deserve.

I go.

✦ ✦ ✦

Kansas means TIG welding at a company called Heatron — I had taken welding as my shop class in high school, which turned out to be transferable —

precision heating elements that go into laboratory equipment and industrial machines, things that have to get hot in exactly the right place at exactly the right temperature. I learn to read a weld, to feel when the current is right, to understand in a way that goes past cognition into reflex what the metal wants to do and how to make it do what you need instead. My hands are useful and the day ends with evidence of it.

September 2005, Texas Hold'em at a local VFW hall — I am good at poker for the same reasons I have been good at every game that rewards attention and the reading of other people, and I sit down at the table expecting a comfortable evening. Her name is Avalon, she is seventeen and I am twenty, and we are both kids with more history than we know how to carry and neither of us has the tools to say so. I fall in love completely, without prior reference, without anything to calibrate against — I have been drawn to people before, but this is different, the first time the word actually fits.

✦ ✦ ✦

November 2005. Grandma Lucas has a stroke.

The college plan does not survive contact with this fact — not because she collapses, but because the capacity that had been underwriting the plan is suddenly occupied with something else entirely: survival, recovery, the long slow work of what comes

after a stroke when you are a woman in your sixties who has smoked two or three packs a day for forty years.

With the scaffold gone, I start taking stock of what else exists — what structure is available to a twenty-year-old in Kansas with a welding certificate and no fixed point — and the military comes back as the most legible option on the table. I walk into a recruiting office.

I take the ASVAB and ace it — the recruiter is visibly excited, tells me I can do any job in the military, and I believe him because the score is the most unambiguous thing anyone has shown me in months. I sit across from him and think carefully about what I am and am not willing to do. Nuclear, biological, chemical warfare involves the possibility of pushing a button that ends a lot of lives at once, and I decide no — not because the button would ever be in front of me, but because the role is built around that possibility, and I cannot align myself with it while rejecting what it is built around. Public Affairs is different: you tell the story of what the Marines are doing and why, words and images rather than ordnance, and I understand this role immediately — not as something learned but as something recognized. In January 2006, twenty years old, I ship to boot camp.

Boot camp is the first institution that has ever asked me to prove what I could do with my body, and my body responds by doing things I did not know it could do — distance runs in the dark, in formation, calling cadence until you become the cadence, until you stop hearing it as words and it becomes rhythm, becomes the sound of your own feet on the ground. The obstacle courses. The swimming. The physical training at hours of the morning that feel less like hours and more like a different country. My hands are rough inside two weeks in a way they have never been, and I lose fifteen pounds and then stop losing weight because there is no more to lose that doesn't come from somewhere essential.

I am named platoon scribe — the recruit who maintains the platoon's official records and correspondence, which means I write and keep track and make sure the paperwork is right. In retrospect, a perfect role: it uses what I have. I also memorize knowledge so that other recruits can recite it — Marine Corps history, chain of command, general orders, the Rifleman's Creed — going through it until it is down to the substrate, below thought, until the words come without the experience of retrieving them. Then I give it to whoever needs it. Not generosity, exactly: if they fail, we fail, and I have spent too much of my life watching preventable failure from the inside.

✦ ✦ ✦

February 2006. Grandma Lucas dies.

I am not with her. I did not say goodbye.

This is the sentence I have lived with for twenty years. I was in boot camp, which means I was exactly where I was supposed to be, which means I have absolutely no grounds for guilt in any rational accounting of the situation — and none of that matters at all. She died and I was not there and I did not say goodbye and I was told at morning formation by a staff sergeant whose face, before he said a word, told me something had actually happened.

I leave my platoon and go home, and I stand up at her funeral and say what I had said to someone once, when they asked what she was like: *she saw me, the real me, the brightness and the sadness, and she loved me big.* That is what I say at her funeral. I don't know if I say anything else. I know I meant that.

Then I go back.

✦ ✦ ✦

During range week at Camp Pendleton there is a conflict with a DI — the details feel important at twenty and less so at forty, but the shape of it is familiar: you were right about the principle and wrong about how you handled it, which is the most expensive way to be right. The outcome is Intensive

Training, which means a seventy-pound dog target fifty yards away, drop to push-ups, back up, retrieve the target, and repeat until someone says stop.

I do it, and on the third or fourth repetition something tears — deep in the groin, a tearing that goes past pain into something that lives below pain, something your body reports in a more primitive register. I hit my knees, I vomit, and I come close enough to blacking out that the afternoon goes gray at the edges.

That evening at mess, I cannot sound off. My voice is there but something in my lower body is not cooperating with the act of standing at attention and producing volume, and Staff Sergeant Rodriguez comes to me with a look that is patient and skeptical in equal measure — he has seen this before.

“This recruit reports,” I say, “that he may have been injured.”

He pulls me out, the Navy medic does the assessment and says, with the bluntness that military medicine is built on: *Dude, that is the worst hernia I have ever seen.* The swelling is approximately the size of a grapefruit. I have been carrying it through mess, through the evening routine, through everything after the range, trying to determine whether it was real enough to report. It was real enough to report. They take me in for surgery the next morning.

The surgery goes well, in the sense that the hernia is repaired. Then a piece of gauze is left at the surgical site, then I develop sepsis, then there is a second surgery to find the gauze and address the infection that has been building since the first one.

A broken recruit — this is a technical status as much as a descriptive one, a recruit who cannot complete training due to injury, held in medical while the body does what it needs to do. There is a ward with a tiny library adjoining it, and there is the debit card they gave me when I enlisted, which accesses my pay and works at the commissary — the government will outfit you with everything strictly required for basic training, and for everything else you buy it yourself. Snacks are not strictly required for basic training. Snacks are, however, something considerable when you have been living on mess hall food for months.

I read everything in the library and then read it again, and I lie in the ward and think about what I am doing with my life, which is a thought that arrives with particular force when you are twenty years old and your body has failed you through no fault of your own and the woman who saw you clearly is dead and you did not say goodbye and you are approximately two months away from being healthy enough to return to where you started.

My grandmother is dead, my body has failed me, and the depression settles in — not dramatic, not announcing itself, coming in through a back door and rearranging the furniture. You stop noticing things that used to interest you. The days are long. The ward has a specific smell — antiseptic and old air and something beneath both — and the light through the window is the same quality every day, which after a while stops feeling like consistency and starts feeling like nothing. You eat the snacks and you read the library again and you wait.

There is a moment in one of those weeks where I start a letter to Grandma Lucas and then stop, and then sit there for a long time not writing. I know she is dead and I know the letter will not be sent, and I do not throw it away. I don't know what I do with it. I think I might have just kept writing in my head, for a while — I had been talking to her for most of a decade and the habit did not immediately stop.

The question arrives: discharge, or recovery platoon. Recovery platoon means months of reconditioning, returning to training status, eventually being slotted back into a new platoon at approximately the point in the curriculum where you left. Discharge means leaving. The discharge is medical — neither honorable nor dishonorable, just the category that applies when the body gives out and there's nothing else to call it — but it means going.

I take the discharge.

✦ ✦ ✦

I am twenty years old in June 2006, freshly discharged from the Marine Corps, and I have no car — I came to Kansas without one and the person who might have lent me something is gone — so I walk everywhere for the next year or so. Grandma Lucas is dead. The woman I am in love with is in Kansas. The novel is not written.

What I have is the habit of reading, a body that has been through something and survived it, and the beginning of a question I don't yet know how to finish.

Three things failed in four months: Grandma Lucas's stroke, which took the scaffold. Grandma Lucas's death, which closed the door. The injury and its aftermath, which removed the fallback. None of these failures was my fault. All of them were real.

I had assembled something — not quite a plan, but a direction, with a sense of what would hold it up — and I had been wrong about what was actually holding it up. I had been treating her presence as a given when it was a gift. She had made a decision, sustained across four years of Kansas summers, to be a fixed point for a boy who did not have many, and when her body failed, that decision did not automatically survive her. It had been her energy, her continuous investment. It was her.

I did not have language for this yet. What I had was the weight of it: three things I had assumed were solid, gone inside four months, and the recognition that I had been wrong about which of them was actually load-bearing — and that being wrong had cost me, and that I did not know how to stop being wrong in the same way again.

I am going to spend twenty years finding out.

Chapter 3: The Classifieds

2006–2009

✦ ✦ ✦

> *Life for me ain't been no crystal stair.*
>
> — Langston Hughes, "Mother to Son"

The man comes in off the street on a Tuesday afternoon — or maybe a Wednesday, the days have the same texture in Overland Park in 2007 — and he is in the grip of something, and what he does to the bathroom at Burger King in Overland Park, Kansas, is not something I am going to describe in detail. I'll say:

it is thorough. I'll say: my team looks at it and looks at me and there is a collective decision, made without words, that this one is not happening.

I do not want to clean it either. I look at the bathroom and I feel exactly what they feel.

I clean it anyway — some combination of the math being simple (someone has to, and I am the one willing to) and the particular stubbornness that has been getting me into and out of things since I was six years old. The bathroom gets cleaned. I walk out, take off the gloves, and go directly to the newspaper — the actual newspaper classifieds, because it is 2007 and that is how this works.

I am looking for the listing with the highest starting salary for the least required experience. That is the only criterion I have. I am twenty-one years old and I have been cleaning bathrooms and flipping burgers and waiting tables for the better part of a year and I am not interested in doing it for another year. I want the number. I want the biggest number adjacent to the least barrier — and what comes back is software engineering.

A company called Centriq will train candidates in C#, SQL, HTML, and CSS over six months and place them in entry-level developer roles. The ad is direct about what it is: a boot camp for overlooked people, a

pipeline into an industry that has not historically cared about credentials as long as you can produce. You take an aptitude test. If you pass, you start.

I leave after my shift, go to the facility, take the test, and sign up before I drive home.

✦ ✦ ✦

There is a year before the bathroom.

Kansas, June 2006. I am back from the Marines with a medical discharge and a hernia repair that has mostly healed and a small scar where the gauze was, which is the souvenir the second surgery left. Avalon is there. The plan for college is gone — Grandma Lucas is gone, the scaffold is gone, the fixed point is gone. What is there instead is a city I do not particularly know and a woman I love and the practical question of how to live.

The first apartment is above Tampico's, a Mexican restaurant in a neat stretch of downtown — a little theater on one side, a small indoor mall on the other, the kind of block that has actual foot traffic and feels like a place. The apartment is two bedrooms, one bath, a galley kitchen just wide enough to work in. I love it. The rent is manageable. Two months in, Avalon accidentally — I use that word as she would have used it, with the weight it carries — overdrafts our shared account. The overdraft cascades: one fee triggers the

next, then the next, until the account is negative by more than either of us has coming in. I cannot make rent.

There is no version of this where I stay with the account gone and the rent unpayable, so I drive back to Arkansas and give it a few weeks while the math sorts itself out. Then Avalon calls and says she misses me, and I drive back.

The new apartment: $250 a month, no private bathroom. The toilet and tub are shared, down the hall. There is no shower — if you want to bathe, you sit in the tub. The apartment is fully furnished, utilities included, and it is a slum. The furniture is from some earlier decade and has the smell of having absorbed everything that happened in it, and the walls are thin because the insulation gave up in some earlier decade. It is not a place you live in. It is a place you survive in, briefly, on your way to the next thing.

The next-door neighbor is Paul. Paul is schizophrenic. Most nights are fine — I have lived around mental illness my whole life and I know how to read the register of a building, know when things are okay and when they are not. But some nights Paul laughs and cries simultaneously, in the same breath, a sound that is not quite either one of the things it contains. An eerie sound. The sound of something trying to be two things at once and not managing to be

either. I have never fully shaken it. I hear it sometimes in quiet — not as a memory exactly, more as a frequency I learned to recognize and cannot unlearn.

✦ ✦ ✦

I sold Kirby vacuums, and I was good at it. I had what the job required — I could read a room, I could build rapport in the first thirty seconds, I could find the thing that mattered to the person across from me and speak to that thing rather than reciting a script. Moderate success. Not enough to get out of the $250-a-month apartment, but enough to eat and keep the lights on and feel like a person with a skill rather than a person without options.

Then they moved me to a DPS role: recruiting salespeople, training them on the product and the process, sending them home for the weekend to sell to their friends and family. The DPS role required me to teach other people how to do the thing I had learned to do, which meant I had to understand it well enough to explain it, which meant I had to break it into components and sequence them and figure out which pieces were load-bearing and which ones were style. I was twenty-one years old. I had no framework for what I was doing. I was just doing it.

I think about that role when I am designing onboarding for a new team, when I am trying to figure out how to transfer the thing I know to someone who

does not yet know it. The sequence matters. The order in which you introduce components determines whether they make sense when they arrive, or whether they arrive as noise. I learned that selling Kirby vacuums at twenty-one. I have applied it in every job I have had since.

✦ ✦ ✦

August 2006. Avalon tells me she is pregnant.

I tell her it is her choice. I tell her also — because she asks me, directly, and I am not capable of lying about this — that I want the child and that I cannot stay if she terminates — a statement of what I know about myself, offered as clearly as I can offer it. She does not take it as a threat. She knows me well enough to understand the difference.

She says she has gone through with it. She says this about a month after we have the conversation.

We break up.

About a month after that — a fall afternoon, a door — there is a necklace hanging on the handle and an envelope slipped underneath. The necklace is small and careful. The letter is honest — more honest than most things she ever gave me. She was scared. She had not gone through with it. She had told me she had because she was scared of what the conversation would cost and could not face it directly, and now she

was telling me the truth because she could not hold the lie. She was scared of losing me. She was carrying our child.

We reconcile. Her parents visit — they drive in from wherever they were, they see the apartment, they see the bathroom down the hall and the furniture from a decade I wasn't alive for, and they are mortified with the specific, controlled mortification of parents who have raised a daughter to expect better than this and are now confronted with the evidence that she has chosen something else. They help us into a better place. Not a good place — a better one, which is its own category.

I do not resent them for the mortification. They were right. The apartment was not a place to raise a child. We needed the better place.

✦ ✦ ✦

April 27th, 2007. University of Kansas Hospital.

Avalon is beautiful and scared. No epidural — everything moves too fast, the labor is not long enough for the timing to work, and then it is too far along, and then it is done. I watch her go through something I cannot help with and can only witness, which is its own kind of experience — to be present for the most significant moment in another person's life and to have no role in it except to be present, to be there, to not leave.

Then the nurse hands him to me.

I have heard people say that the moment you hold your child for the first time is unlike anything else. I had filed that in the category of things people say because the vocabulary for what they mean does not exist in standard issue language. And then I am holding William Fox Anderson, who is twenty minutes old and weighs slightly more than I expected, and I understand that the people were not exaggerating. They were reporting accurately from the other side of an experience for which the vocabulary genuinely does not exist.

I am twenty-one years old. I have never felt more love than this and I have never felt more unequipped for anything than this and both of those things are absolutely true at the same time. Fox opens his eyes and looks at the light and does not yet know what any of it is and I know — not as a resolution but as a fact, the way facts present themselves when there is no possibility of arguing with them — that I am going to figure this out. I do not know how. I know I am.

✦ ✦ ✦

The bathroom at Burger King comes three months before this.

The sequencing matters: I find software engineering before Fox arrives, but only because I can already see what is coming. Avalon is six, maybe seven

months along. I know what the child will need, and I know what cleaning bathrooms provides, and the mathematics of those two things do not resolve in any direction that includes the future he deserves. So I go to the classifieds, and the criterion is the same one it has always been — the highest starting salary adjacent to the least required experience — and this time I know exactly what I am looking for it for.

Six months of training: C# and object-oriented programming, SQL and relational databases, HTML and CSS for the front end. The fundamentals, taught fast, taught practically, taught by people who know that everyone in the room came from somewhere else and is trying to get somewhere different and does not have the luxury of learning slowly. I am good at it immediately in the way I was good at the MUDs — the logic of it is not foreign, the idea that you write instructions and the machine follows them, that precision in the writing produces predictable behavior in the result. I had understood that at fourteen on an SSH client at the public library. Now I was learning to call it by its right names.

The thing about learning a formal discipline when you already have the instinct is that the vocabulary arrives and illuminates what was already there. Object-oriented programming gave me the language for something I had been doing in the MUD room builders — the idea that you define a thing's

properties and behaviors together, that a mob is not just a set of stats but an entity with state and rules for how that state changes. SQL gave me the language for the questions I had been asking in every job before it — how do you describe what you need in a way that the system can answer precisely? The names mattered. Not because the instinct was wrong before the names, but because the names let you communicate the instinct to other people, which is the difference between a skill you can use and a skill you can teach.

I completed the program. I was not the most technically polished in my cohort — not the one who had been programming since high school in the formal sense, not the one whose code was the cleanest. What I was, was the one most willing to ask why. Not to challenge the instructors, not to show what I already knew, but because I genuinely wanted to understand what the thing was actually for. The instructors noticed. I was placed.

✦ ✦ ✦

The first job came through a Kansas state work program that subsidizes salaries for overlooked candidates — paying half for six months to open the door — and it placed me at Civic Plus in Manhattan, Kansas, building digital government solutions: websites and portals for municipalities and counties, the digital layer of local government. The rate was

twenty dollars an hour. My mother had reported $9,800 in income for a family of five during my senior year of high school. I am twenty-two years old, and twenty dollars an hour is a king's ransom — something that was supposed to be behind a locked door, except the door was open and I walked through it.

What made that number significant was not only what it bought, though what it bought was real: food, rent, clothes for Fox, the ability to make a plan that extended further than the current month. What made it significant was what it meant structurally. Every job before it had paid in a register that did not compound — gas station wages, Kirby commissions, burger-flipping pay — each of those covered the present moment and nothing beyond it, and you could work hard and cover the present moment harder, but hard work in those registers does not produce a different future. It produces a slightly more survivable present.

Twenty dollars an hour produced a different register entirely, and I knew it: the possibility of compounding, of savings, of skill development, of a career rather than a succession of jobs. I had gotten there through one aptitude test and six months of training and the luck of an industry that was hiring people it wouldn't have hired in any other decade — I knew how much of it was luck, knew how much the luck had depended on one specific decision, walking

out of a burger franchise bathroom and opening a newspaper — and I did not take any of it for granted. I worked harder than I had worked at anything.

✦ ✦ ✦

Civic Plus opens the next door: the Kansas Department of Transportation, and Geographic Information Systems — GIS applications, the software layer on top of spatial data, maps and road networks and the infrastructure information that keeps a state's transportation system legible to the people who manage it. I am learning, as fast as I can learn, not just the tools but the domain: what the software has to do, who uses it and what they need, the gap between what the current systems provide and what would actually be useful. I am twenty-two years old and already instinctively asking the question that will define my career: not *how do I build this* but *what problem does this actually solve, and for whom?*

The Kansas Department of Transportation leads to DISC — the Department of Information Systems and Communications, the central IT hub for Kansas state government. I am building legislative tracking tools for both chambers: bill progress, participation monitoring, anticipated voting patterns. Real systems with real users who have real needs and real frustrations with what they currently have. The work is not glamorous in the startup sense. It is exacting and

detailed and the margin for error is low because the systems I am building are used by people whose work matters. I find this more interesting than glamour. It has the quality the welding had: there is a standard, and the standard is clear, and meeting it requires something real from you.

The legislative tracking work is where I first encounter a problem I will spend twenty years working on: the gap between what users say they need and what they actually need. Legislators and their staff tell me what they want the system to do. I build exactly that. Then they use it and discover that what they said and what they meant were not the same thing — that the need they described was the surface, and the need underneath it was different, and the only way to find the one underneath was to put something in front of them and watch what they did with it.

The frustration of this, early on, is real — you do the work, you do it correctly, and then the work turns out to have solved the wrong problem.

Later I will understand this as a feature of requirements, not a failure of communication. The need does not always know itself. Users are experts in their domain, not in software — they describe what they experience, not what would resolve it. The job is to hear what is said and also hear what is underneath it, to build toward the real problem even when the

stated problem is pointing somewhere else. I do not have the vocabulary for this yet in 2007 and 2008. But I am beginning, encounter by encounter, to develop the instinct.

I am learning what I will spend twenty years learning: that the quality of software is determined almost entirely by the quality of the questions asked before a line of code is written. Not what can I build, but what needs to exist and why and for whom and what happens when it changes. The framework is not there yet. The instinct is there, accumulating — not in theory but in contact with real problems in real systems for real people who need the thing to work.

✦ ✦ ✦

Through all of this, Avalon and I are fraying — she is the mother of my son and she is not here to tell her side of it, and her side of it is real.

I am working hard. I am learning fast. I am building — professionally, financially, in every direction I know how to build. And I am doing this at twenty-two, twenty-three years old, with a child who is growing fast and a relationship that has been under strain since the $250-a-month apartment, since the shared bathroom, since the pregnancy and the reconciliation and the hospital room and the weight of everything that arrived at once in the year Fox was born.

She is struggling with the adjustment to motherhood in ways I cannot fully see because I am not home enough to see them, and because when I am home I do not have as much grace as I could have. That last sentence costs me something to write, because it is the simplest and most honest account of what happened, and simplicity in honest accounting does not erase complexity — it just locates the weight correctly. I was building the right things for the wrong reasons in some proportion I have never been able to calculate precisely. The right reasons: Fox needed a different life than the one I'd had. The wrong reasons: I was building to prove something to myself, to the gap between what I had been and what I needed to become, and that kind of building does not leave much room for another person's experience.

She deserved more grace. I did not have more to give at the time. I have spent the years since then learning to have more.

✦ ✦ ✦

April 2009. Fox is two years old — twenty-three months, more precisely, because at that age the months are the right unit.

I leave Kansas with Fox and a broken relationship behind me. The relationship is not repaired before I leave. It may not have been repairable. What I know is that I leave with the child, that Fox comes with me,

and that the direction I am pointed is Arkansas — back to where I started, with everything I have learned in three years, with a two-year-old in the back seat and a career that is just barely beginning to compound.

✦ ✦ ✦

There is a principle in Harmonic Design that I call volatility-based decomposition: the idea that you should design your systems around what changes, not around what currently is. The designs that fail are almost always designs built around the current state of things — optimized for the problem as it exists today, brittle in the presence of everything that happens next. The designs that hold are built with the question already embedded: what is going to change here, and have I designed for that change, or have I designed as though change won't happen?

I did not have that in language in 2006. Not in 2007 or 2008 or 2009. What I had was the experience of watching things fail — my parents', Bubba's, Grandma Lucas's, mine — and the instinct, built up over twenty years of living inside failure, that the question underneath all of it was the same one. Not *why did this fail* but *what was I assuming would hold that didn't?*

The bathroom at Burger King was a volatility event in miniature: something unexpected happened, the existing system — my coworkers, the status quo,

the implicit agreement about how things worked — was not designed to absorb it, and what remained was a choice. Accept things as they were, or step outside them and ask the next question.

Every door I walked through in those three years came from that question. Centriq came from it. Civic Plus came from it. The state of Kansas came from it. Not because I had a plan — I did not have a plan, not in the sense of a document or a clear destination. What I had was the reflex, the instinct, the capacity to look at a situation and ask: is this load-bearing, or does it just look load-bearing? What is actually holding this up?

The answer was almost never what it appeared to be.

✦ ✦ ✦

I drove out of Kansas in April 2009 with Fox asleep in his car seat and a career that had started in a six-month training program and grown, in three years, into something that could compound — a skill, a rate, the beginnings of a reputation that was small and local and didn't travel, but real. And a two-year-old who was going to need everything I could build for the next sixteen years.

I had, without knowing I had it, the question that would organize the next two decades: not what is the right answer, but what is the right question? Not how

do I build this, but what is this actually for, and what is going to change, and have I built for the change or against it?

The framework was not invented yet. It was not close to being invented. What was there was the instinct — the particular quality of attention that develops when you grow up inside instability — moves, broken things, a sequence of situations that turned out not to hold. You learn to look for the load-bearing element. You learn to find it, name it, and build around it rather than around what looks solid from the outside.

The rest — the framework, the vocabulary, the twenty years of practice at scale — that is the work of the chapters that follow.

But it started here. In a bathroom in Overland Park, Kansas. With a mop and a newspaper and the simplest possible question: what pays the most for the least barrier, and how fast can I get there?

The question was small. What it opened was not small at all.

Appendices

APPENDIX A

Harmonic Design

Harmonic Design

A Unified Software Engineering Framework

Author: William Christopher Anderson **Date:** April 2026 **Version:** 1.0

✦ ✦ ✦

Executive Summary

Four things are true of every long-lived software system. It has a backend that must evolve. It has an interface that must evolve. It has tests that must survive both kinds of evolution. And it has a project plan that must anticipate, sequence, and resource all

three kinds of work. These four concerns are typically treated as separate disciplines, governed by separate frameworks, owned by separate mental models.

Harmonic Design is the framework that eliminates this separation — and the friction it causes — by unifying all four concerns under a single structural discipline.

Harmonic Design synthesizes four frameworks — Volatility-Based Decomposition (VBD) for backend system architecture, Experience-Based Decomposition (EBD) for interface architecture, Boundary-Driven Testing (BDT) for test strategy, and Project Design (PD) for project planning and execution — into a unified practice governed by a single structural map. That map is drawn once, from the volatility of the problem domain, and read at every layer.

The structural parallel is exact. A Manager in the backend corresponds to an Experience in the interface; both are the primary subject of integration tests (real orchestrator, dependencies mocked) and of E2E tests (full stack); both map to integration milestone work packages in the project plan. An Engine corresponds to a Flow; both are the primary home of unit tests; both map to core work packages where most estimation effort concentrates. A Resource Accessor corresponds to an Interaction; both are exercised in unit tests for translation logic and mocked at the boundary in integration tests; both map to boundary work

packages at integration points with external systems. Utility is Utility everywhere — in the architecture, in the tests, and in the project plan as shared infrastructure work. The roles carry the same responsibilities, the same communication rules, the same test profiles, and the same project planning characteristics at every layer because they isolate the same axes of anticipated change.

The consequence of this coherence is significant. A change in business logic touches one Engine, one Flow, one set of unit tests, and one work package in the project plan — structurally bounded, predictably scoped, estimable from the architecture. A new user journey adds one Experience, one or more Flows, one E2E scenario, and one set of dependency edges in the project network. Testing difficulty at any layer signals the same class of structural problem: a boundary is in the wrong place. Estimation difficulty signals the same thing: if a work package is hard to estimate, the component it represents likely spans multiple volatility axes.

One mental model explains the whole system. One diagnosis applies everywhere. One structural map governs what to build, how to verify it, and how to plan, estimate, schedule, and execute the work.

Harmonic Design synthesizes its constituent frameworks into a unified whole. Each pillar retains its full set of rules, role definitions, and validation

approach — but HD governs how they connect, how they reinforce each other, and how structural decisions at one layer propagate through all four. The result is a framework where volatility-first reasoning, applied consistently across backend, interface, tests, and project planning, produces structural isomorphism and predictable change propagation across every layer.

✦ ✦ ✦

Abstract

Harmonic Design is a unified software engineering framework that applies a single organizing principle — isolating change along its natural axes — consistently across backend system architecture, interface architecture, test strategy, and project planning. HD synthesizes Volatility-Based Decomposition (VBD), Experience-Based Decomposition (EBD), Boundary-Driven Testing (BDT), and Project Design (PD) into a coherent whole: four pillars governed by one structural map, producing one mental model that holds across every layer of a software system and the project that delivers it. This paper describes the organizing principle, the structural isomorphism that connects the four pillars, the communication and state-flow rules that hold universally, and the emergent properties that arise

when all four are practiced together. It offers validation criteria for HD coherence and positions the framework relative to existing methodologies.

✦ ✦ ✦

1. Introduction

1.1 The Problem with Separate Frameworks

Software teams rarely suffer from a lack of frameworks. They suffer from frameworks that do not speak to one another.

A team might apply Domain-Driven Design to their backend, atomic design to their component library, coverage targets to their test strategy, and agile ceremonies to their project management. Each framework is locally sensible. Together, they produce a system where the backend's bounded contexts do not correspond to the frontend's component hierarchy, the test strategy was written by convention rather than derived from the architecture, and the project plan was estimated from user stories rather than from the system's structural dependencies. When something changes, the engineer must simultaneously reason in four different structural languages. The seams between frameworks become the places where coupling hides and where plans fail.

The problem is not that the frameworks are wrong. It is that they answer different questions and produce different structural models. A system built across incompatible models carries hidden translation costs at every layer boundary — including the boundary between the system and the plan to build it.

1.2 The Unifying Principle

Across four frameworks — VBD, EBD, BDT, and PD — developed independently and for different domains, the same insight recurs:

Software should be organized by how it changes, not by what it currently does.

This is the claim of VBD: architect backend components around axes of anticipated change, not around functional domains. It is the claim of EBD: structure interface components around the axes of interface volatility — functional, structural, cross-cutting, environmental — not around screens or technical layers. It is the premise of BDT: tests are not a discipline separate from design; they are a mirror of the structure, and their difficulty is diagnostic evidence about where boundaries are wrong. And it is the foundation of PD: project plans derived from the architecture produce accurate estimates, because the components that constitute the work are the same

components that isolate change, and the dependencies that constrain sequencing are the same dependencies that define communication rules.

The same volatility axes appear at every layer. The same four roles — an orchestrating tier, an execution tier, an external-boundary tier, a cross-cutting tier — emerge at the backend, at the interface, in the test profile of each tier, and in the work package classification of the project plan. The same communication rules — state flows downward, results propagate upward as events, horizontal coordination within a tier is prohibited — hold universally. The same validation mechanism — trace a core scenario through the hierarchy without bypassing communication rules — applies to VBD, EBD, BDT, and PD alike.

This is not coincidence. It is the consequence of asking the same fundamental question at every layer.

1.3 What HD Is

Harmonic Design is the practice of applying volatility-first reasoning coherently, simultaneously, across backend architecture, interface architecture, test strategy, and project planning — producing a system whose layers are structurally isomorphic and whose project plan is derived from the same structural map.

Coherent has a specific meaning here. In physics, coherent waves are in phase — they reinforce each other rather than interfering. In Harmonic Design, the four structural models are in phase: the boundaries drawn for volatility reasons in the backend correspond to the boundaries drawn for volatility reasons in the interface, which correspond to the test scope at each level of the spiral, which correspond to the work package boundaries and dependency edges in the project network. They reinforce each other. A structural insight at one layer informs all layers simultaneously — including the project plan.

Harmonic Design is a framework in its own right. Its contribution is not new roles, communication rules, test types, or estimation formulas — those belong to its constituent pillars. HD's contribution is the structural discipline that connects them: the isomorphism rules, the coherence criteria, the diagnostic signals that cross layers, and the governance model that ensures all four pillars evolve together. Practicing the pillars in isolation produces good results. Practicing them as Harmonic Design produces structural properties that none achieves alone.

1.4 Scope and Relationship to Constituent Frameworks

This paper assumes familiarity with VBD, EBD, BDT, and PD. Readers unfamiliar with any of these frameworks are strongly encouraged to consult the

individual papers before proceeding; this paper does not repeat the detailed role definitions, communication rules, estimation formulas, or worked examples that each contains. Instead, it focuses on the connections between them: the structural isomorphism, the coherent communication model, the project planning isomorphism, and the emergent properties that arise from practicing all four together.

✦ ✦ ✦

2. The Organizing Principle

2.1 Volatility as Architectural Truth

Every design decision is a bet about the future. The question is whether the bet is placed consciously or by default.

Traditional decomposition strategies organize systems by what they do: functional layers, domain boundaries, technical tiers. These are structurally coherent bets about the present. They become structurally incoherent bets about the future when the system begins to evolve — because the things that change together are rarely the things that do the same kind of work.

Volatility-first decomposition inverts the organizing question. Instead of asking "what does this component do?", it asks "how does this component

change, and what causes that change?" Components that change for the same reason — and at the same rate — belong together. Components that change for different reasons belong apart, regardless of their current functional relationship.

This is the organizing principle of VBD, EBD, BDT, and PD. Applied to backend systems, it produces Managers, Engines, Resource Accessors, and Utilities. Applied to interface systems, it produces Experiences, Flows, Interactions, and Utilities. Applied to test strategy, it produces a test pyramid whose levels are determined by structural scope rather than convention. Applied to project planning, it produces work packages whose boundaries match component boundaries, dependencies that mirror architectural dependencies, and estimation targets whose volatility characteristics are already known. The principle is the same. The domain of application differs.

2.2 The Four Universal Axes

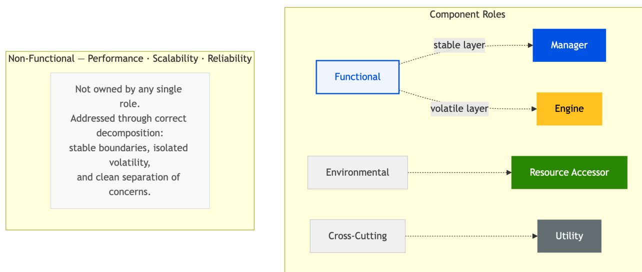
The four axes of volatility that VBD identifies in backend systems appear in equivalent form at the interface layer:

Axis	Backend Driver	Interface Driver
Functional	Business rules, policy, domain logic	User actions, field definitions, validation rules, journey composition, flow ordering, step conditions
Non-Functional	Performance, scalability, reliability	Performance, responsiveness, offline capability
Cross-Cutting	Logging, auth, observability, tracing	Locale, theme, validation conventions, error format
Environmental	Databases, external APIs, infrastructure	Execution platform — rendering environment (React, CLI, WinForms, mobile)

The axes are not identical — the specific pressures that drive functional change in a backend service differ from those that drive change in a form field. But the *type* of change follows the same pattern, and the appropriate structural response is the same: isolate each axis in a dedicated role, enforce communication rules that prevent axis contamination, and let configuration drive composition within each axis.

Figure 1a — Volatility capture in VBD. Functional volatility splits across two tiers: stable orchestration logic lives in the Manager; volatile

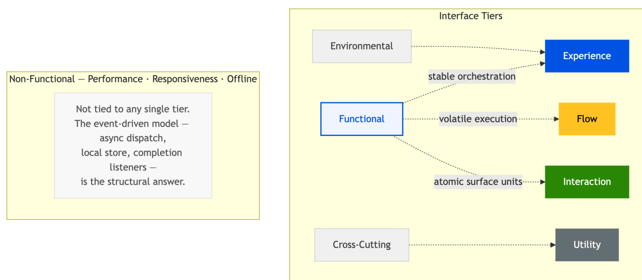
execution logic lives in the Engine. Environmental volatility — external systems, data stores, service APIs — is isolated by Resource Accessors. Cross-cutting capabilities are provided by Utilities. Non-functional volatility (performance, scalability, reliability) is not owned by any single role. It is addressed by the coordinated structure: stable Manager boundaries, isolated Engine volatility, and clean Accessor seams together produce a system whose non-functional properties emerge from correct decomposition.



diagram

Figure 1b — Volatility capture in EBD. The same pattern at the interface layer. Functional volatility spans all three active tiers: the Experience owns the stable journey orchestration, Flows own the volatile goal-directed execution, and Interactions are atomic surface units that can be composed — they have no external awareness and no backend communication. The Experience handles the Environmental axis,

calling the backend API with accumulated state. Cross-cutting capabilities are provided by Utilities. Non-functional volatility (performance, responsiveness, offline capability) is not owned by any single tier. The event-driven structural model — async dispatch, local store, completion listeners — is the structural answer to non-functional requirements.



diagram

✦ ✦ ✦

3. The Structural Isomorphism

3.1 One Map, Four Readings

The central claim of HD is structural isomorphism: the role taxonomy of VBD, the tier taxonomy of EBD, the test scope taxonomy of BDT, and the work package classification of PD map onto each other exactly. They are four readings of the same underlying structural map.

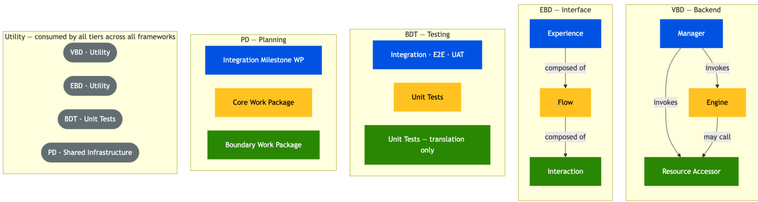


Figure 2 — The Harmonic Design Structural Map: four tiers, each reading identically across VBD, EBD, BDT, and PD. The tiers are structurally isomorphic — the same position in each framework carries the same responsibilities and constraints. Note: VBD and EBD are parallel frameworks, not a call chain. A Manager does not call a Flow; an Experience does not call an Engine. Each framework governs its own layer. The isomorphism is structural, not communicative.

The table form makes the correspondence precise:

Tier	Volatility Axis	VBD Role	EBD Tier	BDT Scope	PD Work Package Type
Orchestration	Functional (stable)	Manager	Experience	Integration · E2E · UAT	Integrati Mileston
Execution	Functional (volatile)	Engine	Flow	Unit	Core Package
External Boundary	Environmental	Resource Accessor	Interaction	Unit (translation only)	Boundar Work Package
Cross-Cutting	Cross-Cutting	Utility	Utility	Unit	Shared Infrastru

> ***Non-Functional note:*** *Performance, scalability, reliability, and responsiveness are not owned by any single tier. They are properties of the coordinated structure — Manager stability, Engine isolation, Accessor seams, and (in EBD) the event-driven model together produce the system's non-functional characteristics.*

> ***BDT note:*** *Integration tests exercise the Orchestration tier — a real Manager or Experience with Execution and External Boundary dependencies mocked. The External Boundary tier's own unit tests cover translation logic only; the tier is mocked at the seam in integration tests and real in E2E.*

> ***PD note:*** *The work package type determines estimation characteristics. Core work packages (Engines/Flows) carry the highest functional volatility and therefore the widest estimation ranges. Integration milestones (Managers/Experiences) are estimated primarily by the number and complexity of seams they coordinate. Boundary work packages (Accessors) are estimated by the complexity of the external system contract. Shared infrastructure (Utilities) is typically estimated with the tightest ranges, as cross-cutting concerns change least frequently.*

3.2 What Isomorphism Means in Practice

Structural isomorphism means that a developer who understands VBD can read the EBD structure without relearning. The role names differ; the structural position and responsibilities are the same. An Experience orchestrates, does not execute, and communicates downward — exactly as a Manager does. A Flow executes one goal's logic, does not coordinate siblings, and communicates results upward — exactly as an Engine does. A Utility is consumed, never coordinates — in both frameworks identically.

More practically, it means that when a developer encounters a structural problem in one layer, they know where to look in the other layers. An Engine with too many responsibilities will have a corresponding Flow with too many responsibilities. A Manager that is doing its own business logic will have a corresponding Experience doing its own API calls. The misalignment propagates coherently — which means fixing it in one layer guides fixing it in the others.

For teams, isomorphism reduces the cognitive load of full-stack development. There is one structural model, read at four levels. A senior engineer does not need to context-switch between "backend thinking," "frontend thinking," "test thinking," and "planning thinking" — they apply the same volatility analysis,

draw the same tier map, and verify the same communication rules. The domain of the code changes; the structural discipline does not.

✦ ✦ ✦

4. The Universal Communication Model

4.1 State Flows Downward; Results Surface as Events

The communication model is identical in VBD and EBD, determines the mock structure in BDT, and defines the dependency edges in PD.

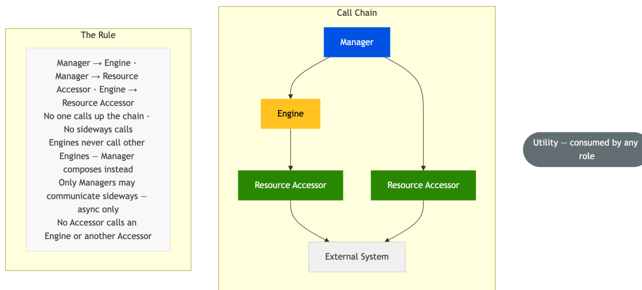
In VBD: orchestration intent flows downward from Manager to Engine to Resource Accessor. Results return upward through function returns and structured response types. Managers do not receive unsolicited signals from Engines; they invoke and receive. Engines do not coordinate with sibling Engines; they are invoked and return. Resource Accessors do not apply business logic; they translate and return. Only Managers emit and consume events; async dispatch, pub/sub, and queued messaging are Manager-layer concerns. Engines and Resource Accessors operate synchronously within a request.

In EBD: configuration drives the Experience, which passes shared state downward to Flows. Flows pass props and callbacks to Interactions. Interactions emit atomic events upward to their parent Flow. Flows

emit completion events upward to the Experience. The Experience holds the accumulated journey state and is the only tier that communicates outward to the backend API.

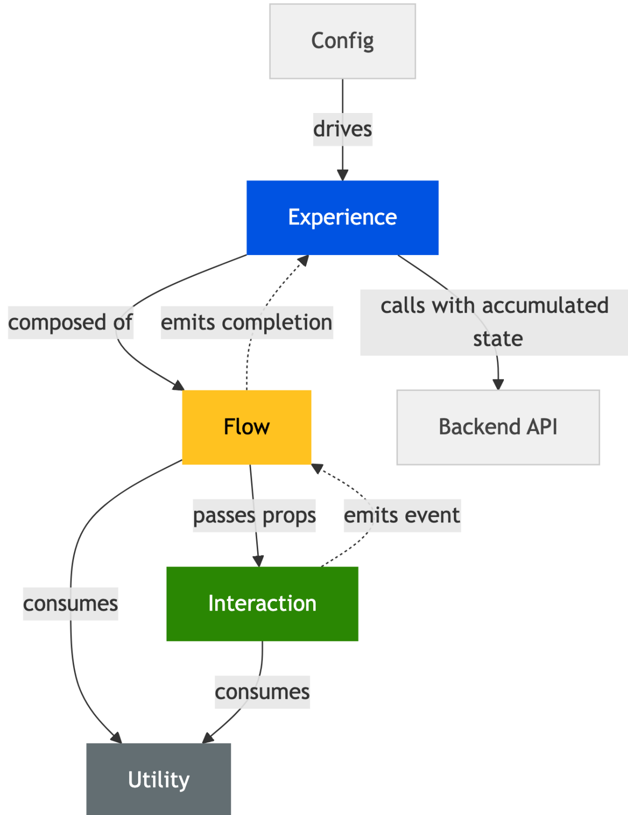
In PD: the same downward flow determines the project network. Manager work packages depend on Engine work packages, which depend on Accessor work packages. The dependency direction in the project network mirrors the communication direction in the architecture. This is not a convention — it is a structural consequence. You cannot build a Manager before the Engines it coordinates exist; you cannot integrate before the components being integrated are complete.

Figure 3a — VBD backend communication: state invoked downward, results returned upward.



diagram

Figure 3b — EBD interface communication: config drives Experience, state passes down, events propagate up. Experience alone calls the backend.



diagram

4.2 The Prohibited Patterns Are the Same

The communication rules in VBD and EBD prohibit the same structural violations, expressed in the terms of their respective domains:

Rule	VBD	EBD
No peer coordination	Engines must not call sibling Engines	Flows must not call sibling Flows
No boundary-skipping	Accessor must not call Engines or peer Accessors	Interactions must not consume other Interactions
No external calls below orchestration	Engines do not call external systems; Accessors do	Flows do not call the backend API; the Experience does
Cross-cutting is consumed	Any role may consume Utilities	Any tier may consume Utilities

These are not parallel analogies. They are the same prohibition — horizontal coordination within a tier, and external coupling below the orchestration tier — applied to the specific roles and artifacts of each layer.

4.3 Consequence for Test Structure

In BDT, where you place mocks follows directly from these communication rules. You mock at the role boundary because the role boundary is where the communication rule enforcement happens. If an Engine must not call sibling Engines, a unit test for

that Engine needs exactly one mock: the Resource Accessor below it. If a Manager must not call peer Managers synchronously, an integration test for a Manager needs all its dependencies mocked — both the Engines and the Resource Accessors it orchestrates. The integration test verifies the Manager's orchestration logic and seam contracts, not the behavior of its dependencies. The test topology is derived from the communication topology. There is nothing to negotiate.

4.4 Consequence for Project Structure

In PD, the communication topology becomes the project network topology. Dependencies between work packages follow the same direction as dependencies between components:

- Utility work packages have no upstream dependencies and form the leaf nodes of the project network.
- Accessor work packages depend on Utility and infrastructure work packages.
- Engine work packages depend on Accessor work packages.
- Manager work packages depend on Engine and Accessor work packages.
- Integration milestones depend on the Manager work packages that compose them.

This ordering is not arbitrary. It is the natural consequence of the communication model: you cannot build what orchestrates before you build what is orchestrated. The project network is derived from the architecture. There is nothing to negotiate here either.

✦ ✦ ✦

5. The Four Frameworks as Pillars

The following sections briefly characterize each framework's core contribution within Harmonic Design. Each framework is documented in full in its individual paper; what matters here is understanding how each pillar carries the same structural weight in the unified practice.

5.1 VBD — The Backend Structural Model

Volatility-Based Decomposition establishes the structural model for backend systems. It identifies four volatility axes and assigns each to a dedicated component role: Managers for non-functional/orchestration concerns, Engines for functional business logic, Resource Accessors for environmental and infrastructural concerns, and Utilities for cross-cutting concerns. It enforces communication rules that prevent volatility contamination across role boundaries and validates the structural model against a small set of core use cases.

VBD's primary contribution to HD: the role taxonomy, the volatility axis framework, and the proof that organizing backend systems by anticipated change produces systems that can absorb that change without widespread refactoring.

The key rule that carries into HD: **boundaries should reflect volatility, not current functional grouping.**

5.2 EBD — The Interface Structural Model

Experience-Based Decomposition applies the same volatility-first logic to interface systems. It establishes three behavioral tiers — Experiences, Flows, and Interactions — plus Utilities, each isolating one of the four interface volatility axes. It enforces the same downward state flow and upward event propagation model. It adds one critical rule with no backend equivalent: the Experience is the exclusive communicator with the backend API. Flows never call the API. The Experience accumulates complete journey state across all its Flows and decides when and what to emit.

One clarification worth stating explicitly: the tier hierarchy describes behavioral scope, not a rigid nesting requirement. An Experience can directly host an Interaction when the scope is genuinely minimal — a single dropdown, a confirmation toggle, a preference that requires no multi-step sequence. The tier exists to

contain a complete goal-directed sequence; when the goal is atomic, no Flow is needed between the Experience and the Interaction. The rule is that Interactions never coordinate laterally or bypass upward — not that a Flow must always be present.

EBD’s primary contribution to HD: the demonstration that the same structural logic that governs backend systems governs interface systems, with the same tier count, the same communication discipline, and the same validation mechanism.

The key rule that carries into HD: **the orchestrating tier (Experience / Manager) is the only tier that crosses the system boundary outward.**

5.3 BDT — The Test Strategy Map

Boundary-Driven Testing maps the test spiral to the structural models defined by VBD and EBD. It establishes that the test type is determined by the structural tier under test: Engines and Flows belong at unit scope, seams between tiers belong at integration scope, complete journeys belong at E2E scope. It treats mock placement as an architectural statement — you mock at boundaries because boundaries are the architecturally significant seams — and reads test difficulty as structural evidence.

BDT’s primary contribution to HD: the demonstration that structural coherence and testing tractability are the same thing. A system with correct

boundaries is easy to test. Testing difficulty is diagnostic of structural problems, not of test strategy problems.

The key rule that carries into HD: **the test spiral is not a testing methodology; it is a structural map. Fix the structure and the tests follow.**

5.4 PD — The Project Planning Model

Project Design derives the project plan from the architectural decomposition. Components become work packages. Dependencies between components become the project network. The critical path through that network determines the project duration. Compression and decompression of that path determine the feasible range of schedules. Risk is quantified objectively from the float distribution across all activities. The result is not a single plan but a set of viable options — typically three — spanning conservative, balanced, and aggressive approaches.

PD establishes several principles that carry directly into HD. The activity inventory is architecture-derived: each component identified during VBD or EBD decomposition becomes a work package with a predictable lifecycle (design, implementation, unit testing, integration). Each pair of connected components generates an integration activity. Each core use case generates a system-level verification activity. The one-to-one rule — each component

assigned to one developer — is a direct consequence of the same boundary discipline that governs the architecture: if a component cannot be built by one developer within the project timeline, it should be decomposed further, and this is an architectural problem, not a staffing problem.

PD's primary contribution to HD: the demonstration that the same structural map that governs what to build and how to verify it also governs how to plan, estimate, schedule, and execute the work. The architecture is the plan.

The key rule that carries into HD: **you cannot estimate what you have not designed. The architecture must precede the estimate, because the architecture defines the work.**

✦ ✦ ✦

6. The Project Design Isomorphism

6.1 The Architecture Is the Plan

The deepest structural connection between PD and the other three pillars is not analogical — it is definitional. In Project Design, the project plan is not constructed from requirements, user stories, or feature lists. It is derived from the architectural decomposition. This means the same structural map that VBD uses to

organize backend components and EBD uses to organize interface components is the map that PD uses to organize the project.

Every VBD component — every Manager, Engine, Resource Accessor, and Utility — becomes a work package. Every EBD component — every Experience, Flow, Interaction, and Utility — becomes a work package. Every dependency arrow in the architecture diagram becomes a precedence relationship in the project network. The critical path through the project is the critical path through the architecture's dependency graph. The architecture is the plan.

This is not a metaphor. When an architect draws a dependency from `SchedulingManager` to `EligibilityEngine`, that arrow means two things simultaneously: at runtime, the Manager invokes the Engine; in the project plan, the Engine work package must complete before the Manager work package can begin integration. The same arrow. Two readings. One structural truth.

6.2 Work Package Classification Follows Tier Classification

The tier classification in VBD and EBD directly determines the characteristics of each work package in PD:

Orchestration tier (Manager / Experience) — Integration Milestone Work Packages. These work packages are primarily integration work. The Manager

or Experience itself contains minimal logic — it orchestrates. Its work package is therefore dominated by integration effort: wiring the seams, verifying the contracts, handling the error propagation paths. Estimation for these work packages focuses on the number and complexity of seams, not on algorithmic complexity. They are typically on or near the critical path because they depend on everything below them.

Execution tier (Engine / Flow) — Core Work Packages. These are where the domain logic lives, and where most estimation effort should concentrate. Engines and Flows carry the highest functional volatility — they embody the business rules and user-facing logic that change most frequently. This volatility translates directly into estimation uncertainty: the work packages for Engines and Flows should carry the widest estimation ranges. They are the components most likely to require design iteration, and the components whose scope is most likely to be affected by requirement changes during execution.

External Boundary tier (Resource Accessor / Interaction) — Boundary Work Packages. These work packages are defined by the complexity of the external system they interface with. The component itself is simple — translation logic only, no business rules. But the work to build and verify it depends on the external system's contract stability, documentation quality, and

availability for integration testing. Estimation for boundary work packages should account for external system risk, which is often the dominant factor.

Cross-Cutting tier (Utility) — Shared Infrastructure Work Packages. Utilities are the most stable tier — they change least frequently, they have no domain-specific knowledge, and they have no upstream dependencies. Their work packages are the most parallelizable and carry the tightest estimation ranges. They form the leaf nodes of the project network, enabling early construction starts.

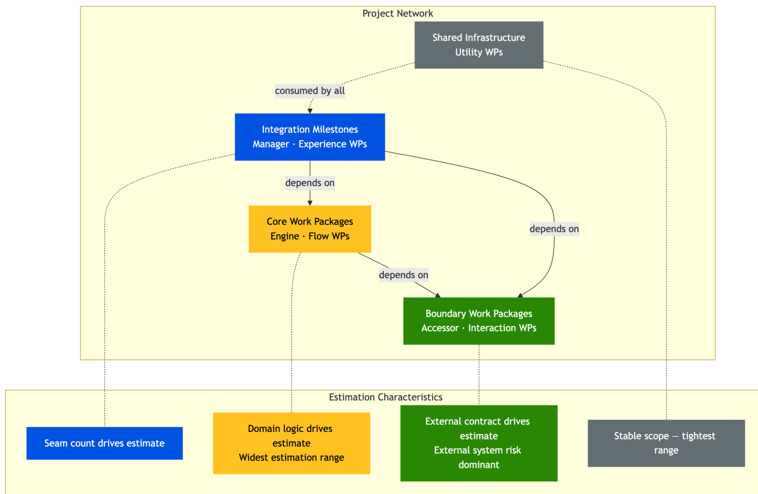


Figure 5 — Work package classification derived from the structural map. Each VBD/EBD tier maps to a PD work package type with distinct estimation characteristics.

6.3 Dependencies Mirror Communication Rules

The communication rules enforced by VBD and EBD — state flows downward, results propagate upward, no peer coordination — produce a specific dependency topology. That same topology is the project network.

In the architecture: Managers invoke Engines; Engines invoke Accessors; Accessors invoke external systems. No Engine invokes a sibling Engine. No Accessor applies business logic.

In the project plan: Manager work packages depend on Engine work packages; Engine work packages depend on Accessor work packages; Accessor work packages depend on infrastructure. No Engine work package depends on a sibling Engine work package (because the components do not communicate). No Accessor work package depends on an Engine work package (because Accessors do not call Engines).

The prohibited patterns in the architecture are the prohibited patterns in the project plan. If an Engine calls a sibling Engine — a communication rule violation — the corresponding project network would show a dependency between peer work packages, creating a cycle or an unnecessary coupling that complicates the critical path. The communication discipline that produces clean architecture also produces clean project networks.

6.4 Risk Maps to Volatility

In PD, risk is quantified objectively from the float distribution across all activities. Activities with zero float are on the critical path and carry maximum risk. Activities with large float can absorb delays.

In HD, the volatility axis framework provides an additional dimension to this risk analysis. The most volatile tiers — Engines and Flows, which carry functional volatility — are the most likely to experience scope change, design iteration, and estimation error during execution. VBD already identifies these components as the functionally volatile tier. PD already identifies the most schedule-sensitive activities through float analysis. When both analyses converge on the same components — when a functionally volatile Engine with a wide estimation range is also on the critical path with zero float — the risk is compound, and the project plan should reflect this.

Conversely, the most stable tiers — Utilities, which carry cross-cutting volatility that changes infrequently — are the safest components on the schedule. Their estimation ranges are tight, their scope is unlikely to change, and their position as leaf nodes in the network means they have the most float. The volatility analysis and the float analysis agree: these are low-risk work packages.

This convergence is not a coincidence. It is a structural consequence of deriving both the architecture and the project plan from the same volatility map. The architecture identifies what changes most. The project plan identifies what is most schedule-sensitive. When both are derived from the same map, the high-volatility components and the high-risk activities are the same things — which means risk mitigation strategies (assigning the best resources, providing wider estimation ranges, investing in design iteration) target the right components automatically.

6.5 Staffing Follows Architecture

PD establishes the one-to-one rule: each component is assigned to one developer. This is a direct application of Conway's Law — the interaction between team members mirrors the interaction between the components they build. A well-decomposed architecture with minimized inter-component coupling naturally minimizes inter-developer communication overhead.

In HD, this principle extends further. Because the architecture already organizes components by volatility tiers, and because volatility tiers map to estimation characteristics and risk profiles, staffing assignments can be made tier-aware:

- The best resources are assigned to critical-path Engine work packages — the most volatile, most uncertain, most estimatively risky components.
- Boundary work packages (Accessors) are assigned to developers with the most experience with the specific external systems involved.
- Utility work packages are assigned flexibly — their tight estimation ranges and high float make them the most forgiving assignments.
- Integration milestone work packages (Managers/ Experiences) are assigned to senior developers who understand the seam contracts — because integration work is primarily coordination work, and coordination requires structural understanding.

Team boundaries align with volatility boundaries, which align with component boundaries, which align with work package boundaries. One structural map governs all four.

✦ ✦ ✦

7. Coherence Properties

When VBD, EBD, BDT, and PD are practiced together as HD — when the same volatility map governs the backend, the interface, the test strategy, and the project plan simultaneously — emergent properties appear that none of the individual frameworks produces in isolation.

7.1 Change Locality Across All Layers

In an HD system, a change in requirements touches exactly one component at each tier, across all layers — including the project plan.

A new pricing rule touches one Engine in the backend. At the interface, the same change might modify one Flow that presents pricing information. In the test suite, the unit tests for that Engine are updated, the unit tests for that Flow are updated, and the integration test for the Manager-to-Engine seam is updated if the contract changes. In the project plan, the change is one new work package, one new dependency edge, one recalculation of the critical path. Everything else is unaffected.

This is stronger than any individual framework achieves alone. VBD localizes backend change. EBD localizes interface change. BDT localizes test change. PD localizes planning change. HD localizes change

across all four simultaneously, because the boundaries at each layer correspond. The blast radius is structurally bounded at every layer, not just one.

The project plan absorbs architectural change the same way the architecture absorbs requirement change. A new Engine means one new work package, one new dependency edge, one recalculation of the critical path. The plan does not need to be replanned from scratch. It absorbs the change structurally, because its structure mirrors the architecture's structure.

7.2 Test Scope Determinism

In an HD system, a developer never has to ask "what kind of test should I write for this?" The answer is structurally determined.

If the component is an Engine or a Flow — execution tier, functional volatility — write unit tests. Mock the Accessor or API boundary below. Assert all contract states.

If the component is a Manager or Experience — orchestration tier — write integration tests at each of its seams. Mock the Engines or Flows it orchestrates. Verify that it correctly handles every state each dependency can emit.

If you are verifying a complete user journey — full stack — write an E2E test. No mocks. Real infrastructure.

The test pyramid is not a recommendation. It is a consequence of the structure. A system with correct HD boundaries produces a correctly shaped pyramid naturally, without any team convention or coverage mandate.

7.3 Estimation Accuracy from Volatility Alignment

In an HD system, estimation targets are volatility-aligned. This means the components being estimated are already classified by their anticipated rate of change, and the estimation approach can be calibrated accordingly.

High-volatility components (Engines, Flows) get wider estimation ranges — because they embody the business rules and user-facing logic most likely to change during execution. These are the components where design iteration is most likely, where scope clarification takes the longest, and where estimation error is historically largest. The architecture already identifies them. The estimation process can account for their uncertainty from the start, rather than discovering it during execution.

Stable components (Managers, Utilities) get tighter estimation ranges — because their scope is well-defined, their logic is minimal (orchestration or cross-cutting), and their change rate is low. The architecture already identifies their stability. The estimation process can trust their scope.

This calibration is not possible when estimating from requirements or user stories, because requirements do not carry volatility information. A user story about “pricing calculation” gives no structural signal about whether the work is an Engine (high volatility, wide range), a Manager (low volatility, tight range), or an Accessor (external system risk). The HD structural map provides that signal before estimation begins.

7.4 Structural Legibility

An HD system has one structural model, not four. Any engineer who understands that model — the four tiers, their volatility axes, their communication rules — can navigate the backend, the interface, the test suite, and the project plan using the same conceptual vocabulary.

This has compounding effects on team organization and onboarding. An engineer learning VBD implicitly learns EBD’s tier structure, because the structural logic is the same. An engineer who understands why Engines do not call sibling Engines automatically understands why Flows do not call sibling Flows. A project manager who understands the dependency graph in the project network automatically understands the communication topology in the architecture — because they are the

same graph. The mental overhead of learning a new layer is reduced to learning the domain-specific vocabulary, not a new structural theory.

7.5 Diagnostic Consistency

Testing difficulty in HD is a single diagnostic, not four separate ones. When something is hard to test at any level — hard to unit test, hard to write integration tests for, hard to write stable E2E tests for — the diagnosis is always the same: a boundary is in the wrong place.

Hard-to-unit-test Engine: it has absorbed Accessor responsibilities, or it is calling a sibling Engine. Hard-to-unit-test Flow: it is making API calls it should not make, or it is aware of sibling Flows. Hard-to-write-integration-test-for Manager: its seams are not well-defined, or its Accessors are embedded rather than injected. Unstable E2E: the Experience does not represent a complete, semantically stable journey.

Hard-to-estimate work package: the component it represents spans multiple volatility axes — an Engine that also does orchestration, an Accessor that also applies business rules. If the component were correctly decomposed, each resulting work package would be estimable in isolation.

The same structural analysis resolves all of these. There is no separate "test problem" or "estimation problem" to debug. There is only a structural problem, and the structural model tells you exactly where it is.

7.6 Risk Quantification from Architecture

In a conventional project, risk is assessed subjectively — experienced practitioners identify likely failure modes and assign probability and impact scores. In an HD system, risk is quantified objectively from two converging sources: the float distribution in the project network (PD) and the volatility classification of each component (VBD/EBD).

Activities with zero float are on the critical path — maximum schedule risk. Components with high functional volatility carry maximum estimation uncertainty. When a zero-float activity corresponds to a high-volatility Engine, the risk is compound and quantifiable. When a high-float activity corresponds to a stable Utility, the risk is minimal and the float can be traded for staffing flexibility.

This convergence means risk mitigation is architectural, not managerial. You do not manage risk by adding buffers to a plan estimated from requirements. You manage risk by ensuring that high-volatility components have adequate float in the project network, that the best resources are assigned to volatile critical-path activities, and that the architecture itself minimizes the number of high-volatility components on the critical path — which is an architectural decision, not a project management decision.

7.7 Configuration-Driven Composition at Every Layer

VBD establishes the principle that configuration should drive composition — that architectural variants should be produced by supplying different configuration, not by writing different code. EBD applies the same principle at the interface layer: an Experience composes different Flows based on the configuration it receives, without changing the Flows themselves.

In an HD system, this principle holds across all layers simultaneously. The same configuration that selects a backend policy variant (a Manager receives different Engine parameters) also drives interface composition (an Experience presents different Flows) and is verified by the same E2E scenario with different configuration inputs. Configuration is a first-class citizen of the architecture, not an afterthought.

✦ ✦ ✦

8. Configuration as a First-Class Citizen

8.1 The Configurability That Correct Structure Enables

Every team wants systems that can be configured rather than recoded. Few achieve it consistently, because configuration-driven behavior requires structural preconditions that are rarely designed for

explicitly. Harmonic Design creates those preconditions as a natural consequence of correct decomposition.

The precondition is simple to state: configuration can only drive behavior when the things that vary are cleanly separated from the things that stay the same. In a system where business rules are interwoven with orchestration logic, where API calls are embedded in flows, where test scaffolding is entangled with component logic — configuration has nowhere to attach. Every configuration variant requires a code change.

In a Harmonic Design system, the structure creates natural attachment points at every tier:

- **Managers** receive configuration that determines which Engines to invoke and in what order — without embedding that logic inside any Engine.
- **Experiences** receive configuration that determines which Flows to compose, in what sequence, under what conditions — without embedding that logic inside any Flow.
- **Engines** and **Flows** receive parameterized inputs that shape their behavior — without knowing where those parameters came from or what else is running alongside them.

The result is a system where new behavioral variants are produced by writing configuration, not code.

8.2 Configuration-Driven Composition in Practice

The most powerful expression of this principle is not configuration that tweaks a parameter — it is configuration that assembles entire systems.

When Managers are stable orchestrators and Engines are stateless executors, a Manager can be driven entirely by a configuration schema: which Engines to invoke, in what order, under what conditions, with what inputs. A new workflow is a new configuration block, not a new codebase.

When Experiences compose Flows based on configuration, and Flows render Interactions based on their own configuration, an entirely new user journey — different steps, different conditions, different validation — can be created from a YAML file. The code is infrastructure. The configuration is the product.

This is not a theoretical capability. It is the practical outcome of structural discipline applied consistently. Teams that practice Harmonic Design find themselves building systems that derive new capability entirely from configuration — and eventually, systems that use configuration to generate *more systems*.

8.3 Software Factories

The most complete expression of configuration-driven Harmonic Design is the software factory: a system whose structural model is stable enough that new instances of that model can be assembled from configuration alone.

A factory built on Harmonic Design principles has the following shape:

- A fixed set of Managers representing the orchestration patterns of the domain. These do not change when new products or workflows are added.
- A fixed set of Engines representing the business rule primitives of the domain. New rules are new Engine configurations, not new Engine classes.
- A fixed set of Experiences and Flows representing the interaction patterns of the domain. New journeys are new Experience configurations, not new interface components.
- A configuration layer — YAML, JSON, a schema, an administrative UI — that assembles these structural elements into new product instances.

The factory itself is a Harmonic Design system. It consumes configuration at its orchestration tier and produces running systems at its output tier. The configuration is the specification; the structural model

is the execution engine. New capability is produced from configuration without touching the factory's codebase.

This is not code generation. Code generation produces new source code from templates. A Harmonic Design factory produces running behavior from structural assembly. Structural assembly is safe, testable, and reversible — because the structural elements being assembled are themselves governed by the same boundary rules, test profiles, and communication contracts as the factory that assembles them.

8.4 Configuration Absorbs Change

The most consequential capability of configuration-driven Harmonic Design is that it changes the relationship between requirements and code. In a conventional system, a change in business requirements produces a change in code. In a configuration-driven Harmonic Design system, many classes of requirement change produce only a change in configuration.

This is not a small distinction. It means:

- A new validation rule is a new configuration entry, not a pull request.

- A new user segment with a different onboarding path is a new Experience configuration, not a new frontend component.
- A new document type that a system must process is a new processing rule configuration, not a new Engine class.
- A new organizational policy that changes how workflows are sequenced is a new Manager configuration, not a refactoring of orchestration logic.

The system absorbs these changes because the structure anticipated them. The volatility that would have been expressed as code churn is instead expressed as configuration variation — contained, reviewable, deployable without a build.

8.5 Configuration-Driven Systems End to End

This principle applies equally at every layer — backend and interface alike. The Harmonic Design system that is fully driven by configuration does not have a configurable frontend and a hardcoded backend. Both layers are configurable, and both are configurable for the same structural reason: the volatile parts (rules, flows, compositions) are separated from the stable parts (orchestration patterns, execution primitives, rendering infrastructure).

Backend: A Manager driven by configuration invokes different Engines in different orders for different tenants, segments, or policy contexts — without a single tenant-specific code path. An Engine parameterized by configuration applies different rule sets to the same computation — without branching on tenant identity inside the rule. A new workflow is a new configuration block. A new policy variant is a new parameter set. The backend codebase does not grow when the business grows; the configuration does.

Interface: An Experience driven by configuration composes different Flows for different user segments, organizational contexts, or product variants — without a single conditional branch in the Experience code. Each Flow renders different Interactions based on its own configuration. The result is an interface that presents entirely different product experiences from the same structural code.

End to end: When both layers are configuration-driven, a new product variant is a new configuration package. Upload it, and both the backend behavior and the interface composition change simultaneously — no code deployment, no build pipeline, no feature flag cleanup. The new variant is immediately available to whatever context the configuration targets.

What this enables in practice:

- **White-labeling** without branching codebases. Configuration supplies branding, behavioral rules, and workflow sequences at both layers. The codebase is shared.
- **Multi-tenant variation** without tenant-specific code. Configuration describes what each tenant sees and how the backend serves them. Both layers compose accordingly.
- **Regulatory compliance variants** without forking. A different compliance rule set is a different configuration block. The Engines that enforce it are the same Engines that enforce all other rule sets.
- **Capability extension by upload.** New rules, new document types, new process definitions, new assets — uploaded as configuration, interpreted by the structural model, immediately operational.
- **Phased capability rollout** without feature flags embedded in logic. A capability is present when its configuration is present. When the configuration is absent, the capability does not appear — no conditionals, no dead code paths.

Neither the backend nor the interface knows which variant it is serving. The backend knows how to orchestrate Engines and invoke Accessors. The interface knows how to compose Experiences,

assemble Flows, and render Interactions. That is the entire codebase. The variation lives in the configuration.

8.6 Dogfooding the Structure

The deepest expression of this pattern is a system that uses its own structural model to extend itself. A Harmonic Design system that produces new workflows from configuration can also define new configuration schemas using the same workflow machinery — because a configuration schema is itself just another workflow, driven by an Experience, composed of Flows, assembled from Interactions.

This is the point at which Harmonic Design becomes more than an architectural methodology: a way of building systems that know how to build more of themselves. The structural model is recursive. The factory can configure new factories. The boundary discipline that makes leaf components testable and replaceable also makes the factory composable at the system level.

Assets, rules, processes, and capabilities can all be uploaded as configuration. The running system interprets them, routes them through the appropriate tier, and makes them available immediately — without a deployment, without a code review, without a build pipeline. The code is stable. The system is not static.

Twenty years of consistent engineering practice produces a consistent observation: the teams that achieve this level of configurability are not the teams with the best technology — they are the teams with the most disciplined boundaries. The factory emerges from the structure. The structure comes from the discipline. The discipline is Harmonic Design.

✦ ✦ ✦

9. How Change Propagates in a Harmonic Design System

The following example traces a single requirement change — adding a tiered user segment to an onboarding system — through all four layers of an HD system. The purpose is to demonstrate that the structural map is the same at every layer, and that the change propagates through corresponding tiers.

The change: The system must now present a different onboarding path for enterprise users versus individual users. Enterprise users require an organizational profile flow; individual users skip it.

9.1 Backend Change (VBD)

A new `SegmentEngine` is added to evaluate user segment from the incoming request. The `OnboardingManager` is updated to call `SegmentEngine` before determining which downstream Engines to

invoke. The `SegmentEngine` is unit tested in isolation — given user attributes, returns the correct segment type. An integration test at the Manager-to-Engine seam verifies that the Manager routes correctly based on each segment value the Engine can return. No other components change.

9.2 Interface Change (EBD)

The `OnboardingExperience` is updated to receive segment state from the backend (already present in the API response) and use it to determine which Flows to compose. An `OrgProfileFlow` is added for enterprise users. The `OnboardingExperience` skips it for individual users. `OrgProfileFlow` is unit tested in isolation — given shared state, steps through Interactions, emits completion. An integration test at the Experience-to-Flow seam verifies that the Experience correctly skips or includes `OrgProfileFlow` based on segment. No existing Flows change.

9.3 Test Change (BDT)

One new unit test suite for `SegmentEngine`. One new unit test suite for `OrgProfileFlow`. Two new integration test paths — one for enterprise, one for individual — at the Manager-to-Engine seam and the Experience-to-Flow seam. Two new E2E scenarios:

enterprise onboarding and individual onboarding, each verifying the complete journey in the real system. No existing tests change.

9.4 Project Plan Change (PD)

One new core work package for `SegmentEngine` (design, implementation, unit testing). One new core work package for `OrgProfileFlow` (design, implementation, unit testing). Two new integration activities at the Manager-to-Engine and Experience-to-Flow seams. The `SegmentEngine` work package depends on the existing Accessor work packages (it reads user attributes). The `OrgProfileFlow` work package depends on the existing Interaction work packages it composes. Both new work packages are inserted into the project network with their dependency edges. The critical path is recalculated. If the new Engine is on the critical path, it receives priority resourcing. If it is not — because it has float via an alternative path — it can be staffed flexibly.

The project plan absorbed the architectural change the same way the architecture absorbed the requirement change: one new component per tier, one new work package per component, recalculate the network. No replanning from scratch. No disconnection between what was designed and what was planned.

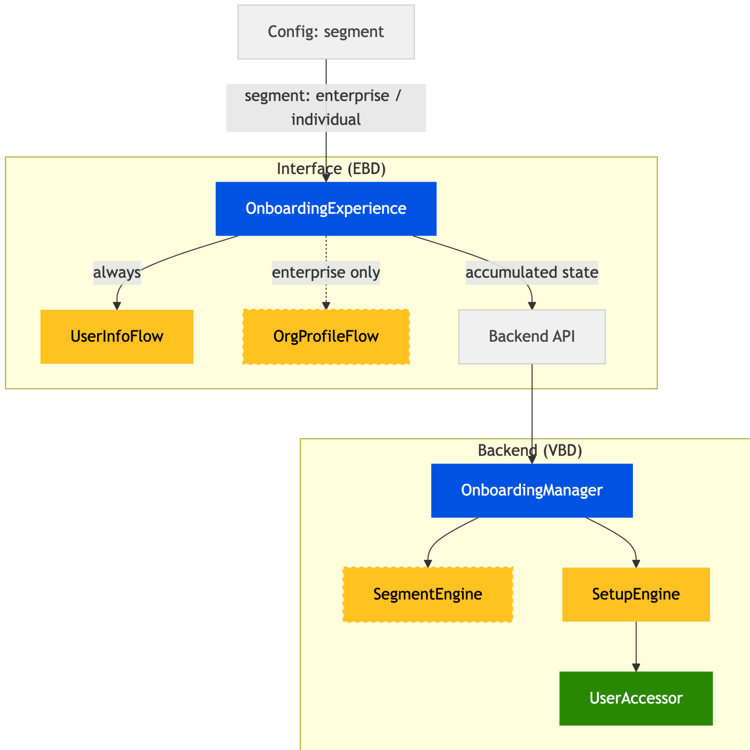


Figure 4 — Change Propagation in an HD System: one new SegmentEngine (dashed, NEW) and one new OrgProfileFlow (dashed, NEW) — one component per tier — propagate through a feature addition without cross-tier or cross-layer spillover. Engines never call other Engines; the Manager routes to each Engine directly.

9.5 What the Propagation Pattern Reveals

The change propagated through exactly one new component per tier, at both the backend and interface layers, with corresponding test additions and a localized update to the project plan. Existing

components did not change. Tests for existing components did not change. The project plan absorbed the change through insertion, not replanning. The structural boundaries contained the change at every layer.

This is not a feature of the specific example. It is the promised behavior of volatility-coherent boundaries: components that change for different reasons are separated, so a change to one reason touches only those components. The feature was added in one Engine, one Flow, and the orchestrating tier that coordinates them — plus the tests that verify each and the work packages that plan each.

A team that did not practice HD would experience the same requirement as a cross-cutting change: logic scattered across multiple Managers, flows embedded in Experience components that now need awareness of segment, tests that can't be scoped because the structure has no stable seams, and a project plan that must be replanned because the work does not decompose into predictable packages. The same requirement, a different structural experience, because the boundaries were drawn differently.

✦ ✦ ✦

10. Validation

10.1 The HD Coherence Test

A system is coherent under HD if the following statements are all true:

Structural:

- Every backend component can be classified as a Manager, Engine, Resource Accessor, or Utility without ambiguity
- Every interface component can be classified as an Experience, Flow, Interaction, or Utility without ambiguity
- The tier hierarchy is consistent: Managers coordinate Engines; Experiences coordinate Flows; no peer coordination exists within a tier
- Utility components in both layers have no domain-specific knowledge and no coordination responsibility

Communication:

- No Engine calls a sibling Engine; no Flow calls a sibling Flow
- No Accessor applies business logic; no Interaction handles flow progression or makes business decisions
- No Flow makes direct backend API calls; only Experiences call the backend

- Any cross-tier communication that appears is justified by an explicit configuration or composition mechanism

Test:

- Every Engine and Flow has a unit test suite; all dependencies are mocked; all contract states are asserted
- Every Manager-to-Engine, Engine-to-Accessor, and Experience-to-Flow seam has integration tests; dependencies are mocked at the outer boundary
- Every core use case / core user journey has an E2E test; no mocks
- Testing difficulty at any level can be traced to a specific structural boundary problem

Project:

- Every component maps to exactly one work package in the project plan
- Every dependency in the architecture maps to a precedence relationship in the project network
- Work package estimation ranges are calibrated to the volatility classification of the corresponding component
- The critical path through the project network corresponds to the critical path through the architectural dependency graph

- Staffing assignments follow volatility boundaries: best resources on volatile critical-path components

Coherence:

- The boundary that separates a Manager from an Engine corresponds to a boundary that separates an Experience from a Flow, a test scope boundary in BDT, and a work package boundary in PD
- A change in business logic touches one Engine, one Flow, the tests for each, and one work package in the plan — not more
- The test pyramid is naturally shaped by the structure, not enforced by coverage policy
- The project plan absorbs architectural changes through insertion and recalculation, not replanning

10.2 Failure Modes and What They Signal

When a system fails the HD coherence test at one layer, it almost always fails at the corresponding position in all layers.

Failure Mode	Structural Signal	HD Diagnosis
Engines calling sibling Engines	Missing Manager for that coordination	Flows will also call sibling Flows; Experience will also be missing a coordination layer
Flows making API calls	Experience is not carrying journey state	Backend Manager is also likely doing its own DB reads to compensate for missing state
Unit tests requiring many mocks	Component spans multiple tiers	The corresponding component in the other layer will have the same smell
E2E tests are the only reliable safety net	Unit-testable execution tier is hollow	Logic has migrated to orchestration or Accessor tier in both backend and interface
UAT surprises not covered by E2E	Core scenario set is incomplete	Structural model does not reflect actual user journeys; Experience composition needs revision
Work packages hard to estimate	Component spans multiple volatility axes	The corresponding component is also hard to test — decompose it
Project plan collapses on requirement change	Architecture does not isolate change	The change touches many components because boundaries are functional, not volatility-based

Failure Mode	Structural Signal	HD Diagnosis
Critical path has many high-volatility items	Architectural dependencies not optimized	Consider restructuring to move volatile Engines off the critical path where possible

The diagnostic is always structural. When testing is painful at one layer, check the corresponding tier in the other layer. When estimation is painful for a work package, check whether the component it represents has a clean volatility classification. The same boundary problem will be present.

✦ ✦ ✦

11. Naming the Practice

11.1 Why This Needs a Name

The four frameworks addressed in this paper — VBD, EBD, BDT, and PD — each have names that position them as domain-specific tools. This is appropriate for learning and for consulting the frameworks individually. But it has an unintended consequence: it encourages teams to treat them as separable options, adopting one and ignoring the others.

The unified practice requires a name that signals that it is unified — that the four frameworks are not a menu but a coherent whole. A team that adopts VBD without EBD will have a well-structured backend and a poorly structured interface; the structural mismatch will surface as friction at the API boundary. A team that adopts EBD without BDT will have a well-structured interface and no principled way to test it. A team that adopts BDT without VBD or EBD will be applying a test strategy to a system whose structure cannot honor it. A team that adopts VBD, EBD, and BDT without PD will have a well-structured, well-tested system — and a project plan estimated from user stories that bears no relationship to the architecture that was so carefully designed.

The four frameworks are effective in isolation. They are designed to work together.

11.2 Harmonic Design

The name is drawn from musical acoustics. A vibrating string produces not one frequency but many simultaneously — a fundamental tone and a series of harmonics above it. The harmonics do not contradict the fundamental; they are expressions of the same underlying physical law at different scales. Each harmonic is complete in itself. Together they produce something richer than any single frequency alone.

VBD, EBD, BDT, and PD are harmonics of the same fundamental: organize by anticipated change. Each is complete in isolation. But practiced together, they produce something qualitatively different from their sum. Structural boundaries reinforce each other across layers. A design decision at one layer resonates into all layers. A structural problem at one layer is audible at all layers. A planning failure signals an architectural misalignment, and an architectural insight simplifies the plan. This mutual reinforcement — the hallmark of harmonic structure — is what the name captures.

With four harmonics, the practice reaches its natural completeness. The fundamental (organize by anticipated change) resonates through what to build (VBD), how to present it (EBD), how to verify it (BDT), and how to plan and execute the work (PD). These four concerns — structure, interface, verification, and delivery — span the full lifecycle of a software system. There is no fifth concern that the fundamental addresses. The four harmonics are complete.

Harmonic rather than *coherent* or *unified* because the word carries the right connotations: resonance, mutual reinforcement, structure that emerges from a shared fundamental rather than being imposed from outside. A harmonic system is not forced into alignment — it finds alignment naturally when all parts are tuned to the same principle.

Design because the practice governs how systems are conceived, structured, verified, planned, and evolved — not just implemented. It applies before the first line of code, persists through all phases of a system's life, and informs decisions across backend, interface, test strategy, and project planning equally.

✦ ✦ ✦

12. Relationship to Existing Practices

12.1 Domain-Driven Design

Domain-Driven Design (Evans, 2003) organizes systems around the conceptual model of the business domain. Bounded contexts define where one model ends and another begins. HD does not conflict with DDD; it adds a dimension to it. Bounded contexts are often natural volatility boundaries — a domain model that changes for one reason is a natural VBD Manager scope. HD can be read as a structural model that operates *within* DDD bounded contexts, governing how each context organizes its own components rather than how contexts relate to each other.

The distinction: DDD draws boundaries between domains. HD draws boundaries within tiers at every layer. Both are necessary in large systems.

12.2 Clean Architecture

Martin's Clean Architecture (2017) organizes systems by dependency direction: high-level policy should not depend on low-level detail. The Dependency Rule — source code dependencies point inward, toward higher-level policy — maps closely onto HD's tier hierarchy: Engines depend on nothing domain-specific, Managers depend on Engine and Accessor interfaces, and Accessors depend on external systems. HD adds the volatility-based *reason* for these dependency directions: Engines are inward because they are the most volatile; Accessors are outward because they isolate environmental volatility.

Harmonic Design and Clean Architecture agree on the communication topology. HD explains *why* that topology is correct.

12.3 Atomic Design

Brad Frost's Atomic Design (2013) provides a vocabulary for hierarchical component composition in interfaces: atoms, molecules, organisms, templates, pages. This addresses structural composition at the *visual* level. EBD — and by extension HD — addresses structural composition at the *behavioral* level: Interactions, Flows, Experiences.

Both taxonomies are valid and non-competing. An Interaction (EBD) might be composed of several atoms and molecules (Atomic Design). An Experience might

correspond to a page template. The two models answer different questions: Atomic Design asks how visual elements compose; EBD asks how behavioral intent composes.

12.4 Test-Driven Development and Behavior-Driven Development

TDD treats tests as a design tool: writing tests before code forces clarity about interfaces and responsibilities. BDD extends this to behavior specifications. HD extends further: the *structure* of the system determines the *structure* of the tests, which in turn validates the *design* of the system. TDD and BDD inform how you write tests at a given level. BDT tells you which level a given component belongs to.

TDD and BDD can be practiced within an HD system without conflict. HD provides the structural scaffolding; TDD/BDD provides the micro-design discipline within each structural tier.

12.5 Critical Path Method and Earned Value Management

CPM (Kelley and Walker, 1959) and EVM (Fleming and Koppelman, 2010) provide the mathematical foundation for Project Design's network analysis, float calculation, and progress tracking. HD does not replace these techniques — PD integrates them. What HD adds is the source of the network: the architectural decomposition. In conventional CPM, the activity list

and dependencies are constructed manually. In HD, they are derived from the architecture. The network is not an estimate of the work — it is a structural consequence of the design.

12.6 Agile Methodologies

Project Design — and by extension HD — is not opposed to agile delivery. The distinction is between planning and execution. PD provides the planning discipline that determines what is built, in what order, with what resources, and at what cost. Agile methods provide the execution discipline that governs how the work is performed day to day. The Feed Me or Kill Me decision maps to Sprint Zero or PI Planning commitment. Sprint velocity maps to the earned value rate. The S-curve maps to the cumulative flow diagram or burnup chart.

Good agile combines architectural discipline with iterative delivery. Bad agile uses agile terminology as a justification for skipping architecture and project design. HD provides the structural foundation that makes agile delivery effective.

✦ ✦ ✦

13. Observations from Practice

The following observations are drawn from applying Harmonic Design across multiple systems over an extended period. They are not prescriptive — they are patterns that have emerged consistently enough to warrant documentation.

13.1 The Structure Reveals Itself Gradually

Teams adopting HD for the first time tend to produce architectures that are partially volatility-aligned. Some components are clean VBD roles; others are hybrids that span two tiers. This is normal and expected. The structural map becomes clearer with each requirement change, because changes reveal which components are truly volatile and which are stable. The first decomposition is a hypothesis. The first six months of requirement changes are the experiment. By the end of the first year, the boundaries that survive are the right boundaries.

13.2 Testing Difficulty Is the Earliest Signal

Before the architecture reveals its weaknesses through requirement changes, the test suite reveals them through testing difficulty. A component that is hard to unit test is almost always misclassified or mis-scoped. Teams that pay attention to testing difficulty as a structural diagnostic — rather than as a complaint

about the testing framework — catch boundary problems months before they manifest as change-locality failures.

13.3 Estimation Accuracy Improves with Structural Maturity

On first application, HD-derived project plans are better than requirement-based estimates but still imperfect — the team has not yet calibrated its estimation ranges to the specific volatility characteristics of its domain. After two or three project cycles, the calibration tightens. Teams learn that their Engines take longer than expected (because functional volatility is underestimated) and their Utilities take less time than expected (because cross-cutting concerns are overestimated). The structural map provides the calibration framework; repeated application provides the calibration data.

13.4 The Plan Absorbs Requirement Changes Without Replanning

This is the observation that most surprises teams new to HD. When a requirement change adds a new Engine and a new Flow, the project plan update is mechanical: insert two work packages, add dependency edges, recalculate the critical path. There is no replanning meeting. There is no argument about scope. The structural map determines what the change

is, and the project network absorbs it. Teams accustomed to requirement changes triggering full replanning sessions find this disorienting at first — and then indispensable.

13.5 Configuration-Driven Systems Compound

Systems built with strict HD boundary discipline tend to become more configurable over time, not less. Each new Engine added with clean boundaries is another Engine that configuration can compose. Each new Flow added with clean state isolation is another Flow that configuration can sequence. The configurability is not designed in — it emerges from the structural discipline. After several years of consistent practice, teams find themselves building new capabilities entirely from configuration, without writing new code. This is not an aspiration; it is a consistent empirical observation.

13.6 Conway's Law Works Both Ways

HD aligns team boundaries with component boundaries, which aligns with volatility boundaries. When this alignment is maintained, Conway's Law becomes an asset: the team structure reinforces the architectural structure. When it is not maintained — when team boundaries cross volatility boundaries — the architectural discipline degrades, because the organizational pressure to coordinate across team

boundaries introduces the same coupling that HD is designed to prevent. Organizational alignment is not optional; it is a structural precondition.

13.7 The Fourth Pillar Changes the Conversation with Management

Before PD, the conversation between engineering and management is adversarial by default: engineering estimates, management negotiates, both sides distrust the outcome. After PD, the conversation is structured: here are three options with quantified cost, schedule, and risk; which do you prefer? The options are not opinions — they are structural consequences of the architecture. Management selects from the feasible zone. Engineering executes the selected plan. The adversarial dynamic is replaced by a decision-making protocol. Teams that adopt PD as part of HD consistently report that the most significant change is not in planning accuracy — it is in the quality of the relationship between engineering and management.

✦ ✦ ✦

14. Conclusion

Software systems fail over time not because of poor initial design, but because change accumulates faster than the architecture can absorb it. The conventional response — adopt a framework, apply it to one layer,

and manage the other layers by convention — does not solve the problem. It solves one layer of the problem and leaves the others unaddressed.

Harmonic Design is the practice of solving all four simultaneously. Not by introducing new concepts, but by recognizing that four independently developed frameworks — VBD, EBD, BDT, and PD — share the same organizing principle, the same structural map, and the same communication rules. They are not separate tools. They are one tool, applied at four layers.

The structural isomorphism is the central insight. A Manager corresponds to an Experience corresponds to E2E scope corresponds to an integration milestone work package. An Engine corresponds to a Flow corresponds to unit scope corresponds to a core work package. A Resource Accessor corresponds to an Interaction corresponds to unit (translation) scope corresponds to a boundary work package. Utility corresponds to Utility everywhere. Once a team sees this correspondence, the four frameworks become one. Learning one teaches the others. A problem in one layer predicts a problem in the corresponding position in the others. A fix at one layer guides the fix at all layers.

The emergent properties of this coherence — change locality across all layers, test scope determinism, estimation accuracy from volatility

alignment, structural legibility, diagnostic consistency, risk quantification from architecture, configuration-driven composition — are not achievable by any individual framework applied to a single layer. They require the full system: backend, interface, test strategy, and project plan organized by the same map, enforcing the same boundaries, communicating by the same rules.

The addition of Project Design as the fourth pillar completes the practice. VBD governs what to build. EBD governs how to present it. BDT governs how to verify it. PD governs how to plan, estimate, schedule, and execute the work. All four are readings of the same structural map. All four are tuned to the same fundamental. The architecture is the plan.

Software engineering has accumulated many frameworks for managing complexity. Harmonic Design is a framework that unifies four of them under a single structural discipline — and demonstrates that practicing them as one produces a kind of structural coherence that none achieves alone.

That coherence is the goal. Harmonic Design is the path.

✦ ✦ ✦

Appendix A: Glossary

- **Boundary Violation** — An instance where a component invokes another in a way that crosses its permitted communication paths, signaling structural misalignment.
- **Change Propagation** — How a modification flows through all four layers when boundaries are correctly aligned.
- **Coherence** — The state where all four pillars reflect the same structural boundaries and the same volatility classifications.
- **Communication Rules** — Constraints on inter-component invocation that hold across all pillars and prevent dependency erosion.
- **Configuration-Driven Composition** — The practice of assembling components through external configuration rather than hardcoded structure, enabling change absorption without code modification.
- **Core Scenario** — A high-level behavior (use case, user journey, test scenario, project milestone) that validates structural decisions at every layer.
- **Diagnostic Signal** — A difficulty indicator (testing difficulty, estimation difficulty, boundary pressure) that reveals structural misalignment.

- **Framework Cross-Reference** — The mapping table showing equivalent concepts across VBD, EBD, BDT, and PD.
- **Harmonic Design** — A unified software engineering framework that applies volatility-first reasoning consistently across backend architecture (VBD), interface architecture (EBD), test strategy (BDT), and project planning (PD), producing structural isomorphism across all four layers.
- **Layer** — One of the four system concerns addressed by HD: backend, interface, tests, project plan.
- **Peer Prohibition** — The universal rule that components at the same tier must not invoke each other directly.
- **Pillar** — One of the four constituent frameworks of Harmonic Design: VBD, EBD, BDT, or PD.
- **Structural Isomorphism** — The property where component roles, communication rules, and volatility boundaries correspond exactly across backend, interface, test, and project layers.
- **Structural Map** — The single decomposition drawn from volatility analysis that governs all four pillars simultaneously.

- **Tier Correspondence** — The mapping between equivalent roles across pillars: Manager?Experience?E2E scope?Integration Milestone WP.
- **Volatility Axis** — A dimension along which change is expected: functional, non-functional, cross-cutting, or environmental.
- **Volatility-First Reasoning** — The practice of using anticipated change as the primary driver of all structural decisions.

✦ ✦ ✦

Appendix B: Framework Cross-Reference

Concept	VBD Term	EBD Term	BDT Term	PD Term
Orchestrating tier	Manager	Experience	E2E / UAT scope	Integration Milestone WP
Execution tier	Engine	Flow	Unit scope	Core Work Package
External boundary tier	Resource Accessor	Interaction	Unit (translation only)	Boundary Work Package
Cross-cutting tier	Utility	Utility	Unit scope	Shared Infrastructure WP
Orchestration rule	MGR coordinates ENG and ACC	EXP coordinates FLW	E2E verifies journey	Milestone depends on core and boundary WPs
Peer prohibition	ENG does not call ENG	FLW does not call FLW	Unit test has one mock boundary	No peer dependency between same-tier WPs
External call ownership	ACC reaches external systems	EXP reaches backend API	Integration tests ACC / API seam	Boundary WP estimated by external contract
Volatility driver	Business rules to Engine	User actions to Interaction	Logic correctness to Unit	

Concept	VBD Term	EBD Term	BDT Term	PD Term
				Widest estimation range to Core WP
Structural driver	Orchestration to Manager	Journey composition to Experience	Collaboration wiring to Integration	Seam count drives milestone estimate
Core validation	Core use cases traced through hierarchy	Core user journeys traced through tiers	Same scenarios at each spiral level	Core scenarios as E2E verification activities
Structural diagnostic	Boundary violation = structural problem	Boundary violation = structural problem	Test difficulty = structural signal	Estimation difficulty = structural signal
Risk source	Functional volatility in Engines	Functional volatility in Flows	Mock complexity at seams	Float distribution across network
Change absorption	New Engine, Manager orchestrates	New Flow, Experience composes	New unit suite, integration path	New WP, recalculate critical path

✦ ✦ ✦

Appendix C: HD Adoption Checklist

A team beginning to adopt Harmonic Design should verify the following at each layer:

Backend (VBD):

- Every component is classified as Manager, Engine, Resource Accessor, or Utility
- Engines do not call sibling Engines or hold workflow logic
- Managers do not embed business rules
- Accessors do not apply policy; they translate and return
- Communication rules are enforced by code review and structural convention

Interface (EBD):

- Every component is classified as Experience, Flow, Interaction, or Utility
- Flows do not call sibling Flows or make direct API calls
- Experiences hold accumulated journey state and are the sole API communicators
- Interactions are atomic; they emit events and receive props
- Utilities have no domain-specific knowledge

Tests (BDT):

- Every Engine and Flow has a unit test suite with all Accessor/API dependencies mocked
- Every Manager-to-Engine, Engine-to-Accessor, and Experience-to-Flow seam has integration tests
- Every core use case and user journey has an E2E scenario
- Test difficulty at any level is treated as a structural signal, not a testing problem

Project Plan (PD):

- Every component maps to exactly one work package
- Dependencies in the project network mirror dependencies in the architecture
- Estimation ranges are calibrated to volatility tier: widest for Engines/Flows, tightest for Utilities
- The critical path is identified and the best resources are assigned to it
- At least three project options (conservative, balanced, aggressive) have been presented to management
- Risk is quantified from float distribution, not from subjective assessment

- Staffing assignments follow the one-to-one rule: one component, one developer
- Staged delivery aligns with the architectural tier hierarchy: Utilities first, Managers last

Coherence:

- Backend tiers correspond to interface tiers (Manager to Experience, Engine to Flow, etc.)
- A change in requirements touches one component per tier, per layer, and one work package in the plan
- The same core scenarios validate structural boundaries in VBD, EBD, BDT, and PD
- The project plan absorbs architectural changes through insertion and recalculation, not replanning
- Estimation difficulty at any work package can be traced to a volatility classification problem in the corresponding component

✦ ✦ ✦

Appendix D: Case Study — Healthcare Scheduling Platform

This appendix presents a fictional but realistic example of Harmonic Design applied end-to-end across all four pillars. The system is a healthcare scheduling platform

that allows patients to book appointments with providers, checking eligibility, searching for available providers, and selecting time slots.

D.1 VBD Decomposition (Backend)

The backend is decomposed into the following components:

Managers: - `SchedulingManager` — orchestrates the complete appointment booking workflow: receive patient request, check eligibility, search providers, reserve time slot, confirm appointment. - `NotificationManager` — orchestrates notification delivery across channels (email, SMS, push) after booking events.

Engines: - `EligibilityEngine` — evaluates whether a patient is eligible for a requested appointment type based on insurance, referral requirements, and plan restrictions. - `ProviderMatchEngine` — matches patients to providers based on specialty, location, network participation, and patient preferences. - `SlotAllocationEngine` — resolves time slot availability against provider calendars, handles conflict detection, and applies scheduling rules (minimum gap between appointments, maximum daily load). - `ConfirmationEngine` — generates confirmation

records, assigns confirmation numbers, and produces the confirmation payload for both patient and provider.

Resource Accessors: - `PatientAccessor` — reads and writes patient records from the patient database. - `ProviderAccessor` — reads provider profiles, credentials, and network participation from the provider directory. - `CalendarAccessor` — reads and writes provider calendar entries from the calendar system. - `InsuranceAccessor` — calls external insurance verification APIs to validate coverage and benefits. - `NotificationAccessor` — sends notifications through external email, SMS, and push notification services.

Utilities: - `DateTimeUtility` — timezone conversions, business hour calculations, recurring schedule generation. - `ValidationUtility` — input validation, sanitization, format verification. - `AuditUtility` — audit trail generation, compliance logging.

D.2 EBD Decomposition (Interface)

Experiences: - `AppointmentBookingExperience` — the complete patient-facing booking journey. Receives configuration, composes Flows, accumulates journey state (selected provider, selected slot, confirmed appointment), and makes the single API call to the backend when the journey is complete.

Flows: - `ProviderSearchFlow` — guides the patient through searching for a provider: enter specialty, enter location, view results, select provider. Emits the selected provider upward to the Experience. - `TimeSlotSelectionFlow` — presents available time slots for the selected provider, allows date navigation, and captures the patient's slot selection. Emits the selected slot upward. - `PatientInfoFlow` — collects or confirms patient information: demographics, insurance, contact details. Emits completed patient info upward. - `ConfirmationFlow` — presents the appointment summary, captures patient confirmation, and displays the confirmation number. Emits confirmation status upward.

Interactions: - `SpecialtyPickerInteraction` — dropdown for selecting medical specialty. - `LocationInputInteraction` — location entry with autocomplete. - `ProviderCardInteraction` — displays a single provider's information; emits selection event. - `CalendarGridInteraction` — renders available time slots in a calendar view; emits slot selection event. - `InsuranceFormInteraction` — insurance information entry fields; emits form completion event. - `ConfirmationSummaryInteraction` — read-only appointment summary display.

Utilities: - `FormValidationUtility` — client-side validation rules. - `FormatUtility` — date, time, phone, and address formatting.

D.3 BDT Test Strategy

The test strategy maps directly to the structural decomposition:

Unit tests: - `EligibilityEngine`: given patient attributes and insurance data (mocked via `InsuranceAccessor`), returns correct eligibility determination for each coverage scenario. - `ProviderMatchEngine`: given patient criteria and provider list (mocked via `ProviderAccessor`), returns correctly ranked and filtered matches. - `SlotAllocationEngine`: given provider calendar (mocked via `CalendarAccessor`), resolves availability correctly including conflict detection and scheduling rules. - `ProviderSearchFlow`: given provider list (via props from Experience), renders search UI, steps through Interactions, emits selected provider. - `TimeSlotSelectionFlow`: given available slots (via props), renders calendar, emits selected slot. - Each Interaction: given props, renders correctly, emits correct events on user action.

Integration tests: - `SchedulingManager` to `EligibilityEngine` seam: real Manager, mocked Engine — verify Manager routes correctly based on every eligibility status the Engine can return (eligible, ineligible, pending-referral, coverage-expired). - `SchedulingManager` to `SlotAllocationEngine` seam: verify Manager handles slot-available, slot-taken, no-slots-available responses. -

`AppointmentBookingExperience` to `ProviderSearchFlow` seam: verify Experience passes correct state and handles Flow completion correctly. - `AppointmentBookingExperience` to `TimeSlotSelectionFlow` seam: verify Experience receives selected slot and transitions to the next Flow.

E2E tests: - Happy path: patient searches for provider, selects slot, confirms appointment — end to end with real infrastructure. - Ineligible patient: patient is denied booking due to insurance restriction — verify the system handles gracefully without partial booking. - No available slots: patient searches, no slots match — verify empty state and alternative suggestions.

D.4 PD Project Plan

The project plan is derived directly from the architecture:

Work Packages (derived from components):

Work Package	Type	Depends On
DateTimeUtility	Shared Infrastructure	—
ValidationUtility	Shared Infrastructure	—
AuditUtility	Shared Infrastructure	—
PatientAccessor	Boundary	DateTimeUtility
ProviderAccessor	Boundary	ValidationUtility
CalendarAccessor	Boundary	DateTimeUtility
InsuranceAccessor	Boundary	ValidationUtility
NotificationAccessor	Boundary	—
EligibilityEngine	Core	PatientAccessor, InsuranceAccessor
ProviderMatchEngine	Core	ProviderAccessor
SlotAllocationEngine	Core	CalendarAccessor
ConfirmationEngine	Core	PatientAccessor, CalendarAccessor
SchedulingManager	Integration Milestone	EligibilityEngine, ProviderMatchEngine, SlotAllocationEngine, ConfirmationEngine
NotificationManager	Integration Milestone	ConfirmationEngine, NotificationAccessor
FormValidationUtility (UI)	Shared Infrastructure	—

Work Package	Type	Depends On
FormatUtility (UI)	Shared Infrastructure	—
SpecialtyPickerInteraction	Core (UI)	FormValidationUtility
LocationInputInteraction	Core (UI)	FormValidationUtility
ProviderCardInteraction	Core (UI)	FormatUtility
CalendarGridInteraction	Core (UI)	FormatUtility, DateTimeUtilit
InsuranceFormInteraction	Core (UI)	FormValidationUtility
ConfirmationSummaryInteraction	Core (UI)	FormatUtility
ProviderSearchFlow	Core (UI)	SpecialtyPickerInteraction, LocationInputInteraction, ProviderCardInteraction
TimeSlotSelectionFlow	Core (UI)	CalendarGridInteraction
PatientInfoFlow	Core (UI)	InsuranceFormInteraction
ConfirmationFlow	Core (UI)	ConfirmationSummaryIntera
AppointmentBookingExperience	Integration Milestone (UI)	ProviderSearchFlow, TimeSlotSelectionFlow, PatientInfoFlow, ConfirmationFlow
Backend-UI Integration	Integration Milestone	SchedulingManager, AppointmentBookingExperie

Critical Path: DateTimeUtility (5) -> CalendarAccessor (10) -> SlotAllocationEngine (15) -> SchedulingManager (15) -> Backend-UI Integration (10) = **55 days**

Parallel paths through InsuranceAccessor (15) -> EligibilityEngine (15) = 30 days from utility start, with 10 days of float relative to the critical path.

The UI critical path runs: FormValidationUtility (5) -> InsuranceFormInteraction (5) -> PatientInfoFlow (10) -> AppointmentBookingExperience (15) -> Backend-UI Integration (10) = 45 days. This path has 10 days of float relative to the backend critical path.

Three Options:

Conservative (70 days, lowest risk): All-normal durations plus 15-day decompression buffer. Risk index approximately 0.35. Staffing: 4 developers peak.

Balanced (55 days, moderate risk): All-normal durations, no buffer. Risk index approximately 0.50. Staffing: 6 developers peak. Best resources assigned to SlotAllocationEngine and EligibilityEngine (highest functional volatility, critical path or near-critical).

Aggressive (42 days, highest risk): Critical-path compression on SlotAllocationEngine (15 to 10 days) and SchedulingManager (15 to 10 days). Network restructuring to enable parallel Engine development against Accessor interfaces. Risk index approximately 0.70. Staffing: 8 developers peak. Requires top resources on all critical-path activities.

D.5 Mid-Project Requirement Change: Adding Telehealth

Midway through construction, a new requirement arrives: the platform must support telehealth appointments in addition to in-person visits. Patients should be able to choose between telehealth and in-person when booking.

How the change propagates through all four layers:

VBD (Backend): No new backend components are required. Telehealth is not a backend architectural concern — it is a visit type. The `SchedulingManager` books the appointment as requested; the resulting visit record carries a `visitType` field (in-person or telehealth) along with provider contact details. The backend has no awareness of video conferencing, room configuration, or connection logic. No existing Engines change.

EBD (Interface): Two new components. A `VisitTypeFlow` is added, positioned before `ProviderSearchFlow` in the `AppointmentBookingExperience`, presenting the patient with a choice between in-person and telehealth. A `TelehealthInteraction` renders the telehealth connection interface once the Experience receives back a confirmed visit with `visitType: telehealth` — it uses the provider details in the visit record to connect the patient to the appropriate telehealth platform. The Experience uses the visit type to configure

downstream Flows — `ProviderSearchFlow` filters by telehealth-capable providers when applicable. No existing Flows change.

BDT (Tests): New unit tests: `VisitTypeFlow` (renders visit type selection, emits choice), `TelehealthInteraction` (given a confirmed telehealth visit record, renders connection UI). New integration test: `AppointmentBookingExperience` to `VisitTypeFlow` seam. New E2E scenario: patient books a telehealth appointment end to end. No backend tests change — the backend did not change.

PD (Project Plan): Two new UI work packages: `VisitTypeFlow` (core UI, 5 days, depends on `VisitTypeInteraction`), `TelehealthInteraction` (boundary UI, 5 days). No new backend work packages — the backend did not change. Dependency edges are inserted into the interface project network. The critical path is recalculated.

The structural map contained the change at every layer. Two new interface components; no new backend components. Existing components absorbed the change through configuration. The plan absorbed the change through insertion and recalculation. No replanning session was needed. No existing code was modified — only the Experience's Flow composition configuration.

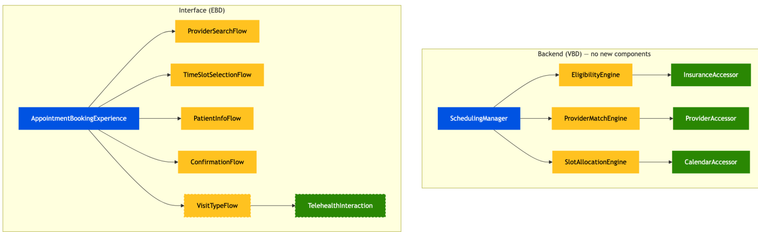


Figure 6 — Telehealth change propagation: two new interface components (dashed border — VisitTypeFlow and TelehealthInteraction). No new backend components — telehealth is a frontend concern. The backend returns a visit record with type and provider details; the frontend handles the connection.

✦ ✦ ✦

References and Influences

William Christopher Anderson Anderson, William Christopher. *Volatility-Based Decomposition in Software Architecture*. February 2026. Anderson, William Christopher. *Experience-Based Decomposition*. March 2026. Anderson, William Christopher. *Boundary-Driven Testing*. March 2026. Anderson, William Christopher. *The Design of Projects: A Practitioner-Oriented Articulation*. February 2026.

Harmonic Design synthesizes these four frameworks. The role taxonomies, volatility axes, communication rules, estimation methodologies, and validation mechanisms in this paper are taken from these sources. HD's contribution is the recognition of

their structural isomorphism and the articulation of the coherence properties that emerge from practicing all four together.

Juval Lowy Lowy, Juval. *Righting Software.* Addison-Wesley, 2019.

The Manager-Engine-Resource Accessor taxonomy that underlies VBD — and by extension the orchestration-execution-accessor tier structure that appears in all four HD frameworks — originates in Lowy's IDesign methodology. The project design methodology that derives project plans from architectural decomposition also originates in IDesign. HD extends IDesign's backend structural model across the interface, test, and project planning layers.

David L. Parnas Parnas, David L. "On the Criteria To Be Used in Decomposing Systems into Modules." *Communications of the ACM*, 1972.

Parnas's information hiding principle — that modules should be designed around design decisions likely to change — is the intellectual foundation of volatility-first decomposition. HD is the system-scale extension of this principle, applied simultaneously at the backend, interface, test strategy, and project planning levels.

Eric Evans Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley, 2003.

DDD's bounded context concept and HD's tier boundaries are complementary rather than competing. HD operates within bounded contexts, governing how each context organizes its internal structure.

Robert C. Martin Martin, Robert C. *Clean Architecture*. Pearson, 2017.

Martin's Dependency Rule and boundary discipline align with HD's communication model. HD provides the volatility-based account of why those dependency directions are correct.

Martin Fowler Fowler, Martin. "The Practical Test Pyramid." *martinfowler.com*, 2018.

BDT builds on Fowler's test pyramid by grounding its levels in structural tiers. HD extends this further by making the test pyramid a consequence of the structural isomorphism across all four layers.

James E. Kelley and Morgan R. Walker Kelley, James E.; Walker, Morgan R. "Critical-Path Planning and Scheduling." *Proceedings of the Eastern Joint Computer Conference*, 1959.

The Critical Path Method provides the mathematical foundation for the network analysis, float calculation, and critical path identification used in Project Design and, by extension, in the HD project planning layer.

Quentin W. Fleming and Joel M. Koppelman Fleming, Quentin W.; Koppelman, Joel M. *Earned Value Project Management.* Fourth Edition. Project Management Institute, 2010.

Earned value management provides the tracking and projection techniques used during project execution in the PD pillar of HD.

Frederick P. Brooks Jr. Brooks, Frederick P. *The Mythical Man-Month.* Addison-Wesley, 1975.

Brooks's observation that adding people to a late project makes it later is reflected in PD's treatment of staffing, compression limits, and the one-to-one rule that aligns with HD's component boundaries.

✦ ✦ ✦

Author's Note

Harmonic Design as a unified discipline is the author's original work. The specific recognition that Volatility-Based Decomposition, Experience-Based Decomposition, Boundary-Driven Testing, and Project Design are harmonics of the same fundamental principle — that their structural tiers are isomorphic; that practicing all four together produces properties none achieves alone; that the architecture is the plan; and that this coherence is what enables the configuration-driven capability described in Section 8 — has not, to the author's knowledge, been articulated

before. After twenty years of practicing this way across many systems and organizations, the author has not encountered another practitioner who applies all four as a unified discipline, or who describes the resulting configurability and planning predictability as structural consequences rather than architectural features.

The four pillars have different origins.

Volatility-Based Decomposition and **Project Design** are deep articulations of Juval Löwy's IDesign methodology — the Manager/Engine/Resource Accessor taxonomy originates there, as does the practice of deriving project plans directly from architectural decomposition. The author's contribution is years of applied refinement and the written articulation of these ideas.

Experience-Based Decomposition builds on Alan Cooper's goal-directed design, Brad Frost's Atomic Design, and VBD's volatility-first thinking. The specific synthesis into a structured tier hierarchy with explicit communication rules, configuration-driven composition, and deliberate isomorphism with the backend component taxonomy is the author's own; if prior art exists under a different name, the author has not encountered it.

Boundary-Driven Testing is the natural consequence of applying VBD and EBD correctly, made explicit. The framework that derives each

component's test profile from its architectural role, reads mock placement as boundary evidence, and treats testing difficulty as a structural diagnostic rather than a tooling problem is, to the author's knowledge, not articulated this way elsewhere.

The configurability properties described in Section 8 are not theoretical. The author has built production systems driven entirely by configuration — backends that derive complete workflow behavior from schema, interfaces that compose from YAML, and factories that extend their own capability through configuration uploads without code deployment. The project planning properties described in Section 6 are equally empirical — the author has planned and delivered projects where the project network was derived from the architectural decomposition, where requirement changes were absorbed through work package insertion rather than replanning, and where management selected from three quantified options rather than negotiating a single estimate. These outcomes were not engineered in as features. They emerged from the structural discipline described in this paper, applied consistently over time.

The four pillars govern the complete lifecycle: VBD governs what to build, EBD governs how to present it, BDT governs how to verify it, and PD governs how to plan and execute the work. With all four in place, the structural map drawn from volatility

analysis governs every phase of a software system's lifecycle — from initial decomposition through project planning, implementation, testing, and ongoing evolution. The architecture is the plan. The plan is the architecture. They are one map, read four ways.

Harmonic Design is a framework, but not in the sense of a tool to be picked up and applied mechanically. It is a framework in the sense of a structural discipline — one that compounds over time, producing qualitatively different kinds of systems the longer it is applied with consistency.

✦ ✦ ✦

Distribution Note

This document is provided for informational and educational purposes. It may be shared internally within organizations, used as a reference in architectural and design discussions, or adapted for non-commercial educational use with appropriate attribution. All examples are generalized and abstracted to avoid disclosure of proprietary or sensitive information.

APPENDIX B

Volatility-Based Decomposition

Volatility-Based Decomposition (VBD) in Software Architecture

Organizing Architecture Around Anticipated Change: A Practitioner-Oriented Articulation

Author: William Christopher Anderson **Date:** April 2026 **Version:** 1.0

✦ ✦ ✦

Executive Summary

Modern software systems rarely fail because of poor initial design; they fail because change accumulates faster than the architecture can absorb it. Volatility-

Based Decomposition (VBD) addresses this problem by treating change as the primary organizing force in system design.

Rather than decomposing systems solely by domain concepts or technical layers, VBD organizes architectural boundaries around anticipated sources of volatility — functional change, non-functional pressures, cross-cutting concerns, and environmental dependencies. By aligning components with these forces, systems can evolve without widespread refactoring.

At a practical level, VBD applies established component roles:

- **Managers** coordinate workflow and intent and remain stable over time.
- **Engines** encapsulate business rules, computation, and transformation logic that change more frequently.
- **Resource Accessors** isolate interactions with databases, external services, vendors, and infrastructure.
- **Utilities** encapsulate cross-cutting capabilities such as logging, monitoring, security, and observability, allowing these concerns to evolve independently without contaminating core business logic.

These roles are reinforced through explicit communication rules and validated against a small number of core use cases. Over time, this approach localizes change, reduces unintended coupling, and preserves architectural integrity even as systems and organizations grow.

VBD is most effective in long-lived systems, platform architectures, and integration-heavy environments where change is constant and unavoidable. It provides architects and senior engineers with a clear, practical reference for applying volatility-first architectural thinking at scale.

✦ ✦ ✦

Abstract

Modern software systems operate in environments defined by continuous change — shifting business requirements, evolving user expectations, regulatory pressure, and rapid technological advancement. Traditional architectural decomposition techniques often fail to account explicitly for change as a primary design force, resulting in brittle systems that degrade over time. Volatility-Based Decomposition (VBD) is an architectural approach that treats change as a first-class concern by identifying, classifying, and isolating areas of anticipated volatility within a system. This paper presents a structured articulation of Volatility-

Based Decomposition, covering functional and non-functional volatility, cross-cutting concerns, core use cases, and component role definition. By aligning architectural boundaries with volatility axes, VBD supports the design of flexible, maintainable, and evolvable software systems. The articulation emphasizes modularity, explicit communication rules, and continuous architectural evaluation, providing architects and senior engineers with a practical reference for applying VBD consistently in long-lived systems.

✦ ✦ ✦

1. Introduction

Software architecture exists to manage complexity over time. While many architectural approaches focus on current functional requirements, the dominant force acting on long-lived systems is change. Business models evolve, regulations shift, infrastructure platforms are replaced, and user expectations rise. Architectures that do not explicitly account for these forces tend to accumulate coupling, resist modification, and require costly refactoring or replacement.

Volatility-Based Decomposition (VBD) addresses this problem by treating change — not functionality — as the primary driver of architectural structure. Rather

than decomposing systems solely by domain concepts or technical layers, VBD decomposes systems along axes of anticipated volatility. This approach enables architects to localize the impact of change, reduce unintended coupling, and preserve system integrity as requirements evolve.

This paper provides a practitioner-oriented articulation of VBD. It describes how to identify volatility, align components to volatility boundaries, apply established component roles and communication rules, and validate architectural decisions over time.

Business strategy evolves continuously. Markets shift. Regulations change. Competitive pressures emerge without warning. To manage this complexity, organizations naturally structure themselves around functional responsibilities such as Sales, Operations, Finance, and Compliance. This functional decomposition brings clarity of ownership, accountability, and decision-making authority.

Software systems frequently mirror this structure. Teams, codebases, and modules are aligned to functional domains in an attempt to reflect how the business operates. Early in a system's life, this alignment can be effective, as changes tend to be localized and coordination costs remain manageable.

Over time, however, tension emerges. The business adapts quickly, while software systems resist change. Most meaningful changes cut across functional boundaries rather than remaining contained within them. Enhancements require coordination across teams and components. Small adjustments trigger broad testing cycles. Risk increases as more parts of the system must move together.

This divergence exposes a fundamental mismatch: while organizations decompose for accountability, volatility does not respect functional boundaries. Volatility-Based Decomposition addresses this mismatch by aligning architectural boundaries with change dynamics rather than organizational structure alone.

✦ ✦ ✦

2. Background and Architectural Foundations

Volatility-Based Decomposition builds upon established architectural principles, including separation of concerns, information hiding, modularity, and loose coupling. Classic decomposition strategies — such as layered architectures, service-oriented architectures, and microservices — implicitly attempt to manage change by isolating responsibilities. However, these approaches often rely on static assumptions about where change will occur.

VBD makes these assumptions explicit. It acknowledges that not all parts of a system change at the same rate or for the same reasons. By identifying which aspects of a system are most likely to change, architects can proactively structure boundaries that align with those forces, rather than reacting after change has already caused architectural erosion.

✦ ✦ ✦

3. Defining Volatility in Software Systems

For the purposes of this paper, volatility is defined as the likelihood that a given system responsibility, requirement, or implementation detail will change over time, along with the frequency and impact of that change.

3.1 Functional Volatility

Functional volatility refers to changes in system behavior driven by evolving business needs, user feedback, or regulatory requirements. Examples include the addition of new features, modification of existing workflows, or removal of obsolete functionality. Functional volatility is most commonly associated with core use cases and domain logic.

3.2 Non-Functional Volatility

Non-functional volatility concerns changes to system qualities such as performance, scalability, reliability, security, and maintainability. These changes are often driven by external forces, including infrastructure upgrades, platform migrations, or increased usage demands.

3.3 Cross-Cutting Volatility

Cross-cutting concerns — such as logging, monitoring, authentication, authorization, and error handling — exhibit volatility that spans multiple components. Changes to these concerns can have widespread impact if not properly isolated.

3.4 Environmental and Infrastructure Volatility

Infrastructure platforms, third-party services, deployment models, and hosting environments are subject to frequent change. Treating these elements as stable foundations often leads to tight coupling between business logic and infrastructure details.

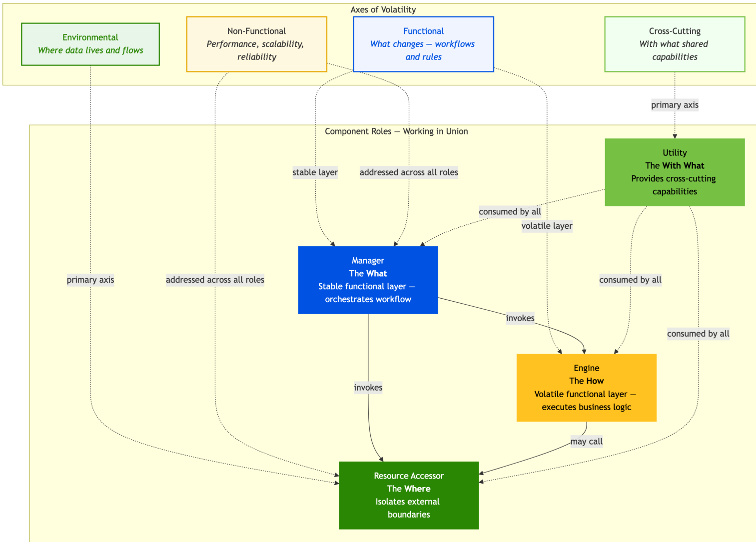


Figure 1 — The Four Axes of Volatility and the Component Roles That Encapsulate Them: each role addresses a primary axis of volatility, but the roles work in union — the Manager orchestrates the what, the Engine executes the how, the Resource Accessor isolates the where, and the Utility provides the with-what. No single role contains all change; the four together localize change across every axis.

✦ ✦ ✦

4. Identifying Core Use Cases and Volatility Axes

The first step in volatility analysis is identifying core use cases — the high-level behaviors that define the system's purpose. Core use cases provide the context necessary to evaluate where change is most likely to occur.

The term *core use case* is intentionally narrow. In practice, even very large organizations tend to have a surprisingly small number of truly core use cases — often fewer than five. These represent the fundamental value-producing behaviors of the system, not the many procedural variations that surround them.

While business analysts may document dozens or even hundreds of use cases, the majority are not architecturally core. They are alternative paths, conditional flows, exception handling, or policy-driven variants of a much smaller set of essential behaviors. Treating all documented use cases as equal drivers of architecture obscures volatility rather than revealing it.

Architects should identify volatility axes by:

- Reviewing existing requirements, user stories, and architectural documentation
- Interviewing stakeholders across business, engineering, and operations
- Analyzing historical change patterns in similar systems
- Considering plausible future business and technology shifts

The goal is not to predict exact changes, but to identify where change is likely and why.

✦ ✦ ✦

5. Volatility-Based Decomposition

Volatility-Based Decomposition proceeds through the following steps:

1. Identify core use cases that represent the system's primary value.
2. Enumerate volatility axes across functional, non-functional, cross-cutting, and environmental dimensions.
3. Classify responsibilities based on their likelihood and drivers of change.
4. Define architectural boundaries that align with volatility classifications.
5. Apply established component roles and communication rules to isolate volatile responsibilities from stable ones.

This process results in an architecture where change is localized and predictable, reducing the risk of cascading modifications.

✦ ✦ ✦

6. Component Roles and Communication Rules

Clear component roles and communication rules are essential to preserving volatility boundaries. In addition to Managers, Engines, and Resource

Accessors, Volatility-Based Decomposition explicitly recognizes Utilities as a first-class role for isolating cross-cutting volatility. The roles and rules described in this section are derived from Juval Löwy's IDesign methodology and component-oriented architecture teachings.

One aspect of the methodology that is often undervalued is the explicit separation of business logic into two distinct concerns: orchestration and execution. Orchestration governs workflow sequencing, coordination, and intent, while execution encapsulates the business rules and policies that perform the work.

When orchestration logic and execution logic are interwoven within the same unit, they become change-coupled. A modification to workflow sequencing can force changes in business rule implementation, and a modification to business rules can require restructuring the workflow. This tight coupling increases fragility by expanding the blast radius of change and reducing predictability.

By separating orchestration from execution, architectures can absorb these changes independently, allowing workflows and business rules to evolve at different rates without destabilizing the system.

A useful mental model for classifying components is to ask **what**, **how**, and **where**:

Role	Question	Concern
Manager	**What** does the system do?	Orchestration — workflow, sequencing, intent
Engine	**How** does it do it?	Execution — business rules, calculations, policies
Resource Accessor	**Where** does data live?	Integration — databases, vendors, external systems
Utility	**With what support?**	Cross-cutting — logging, auth, monitoring, observability

If a unit of work decides *what* happens next, it belongs in a Manager. If it computes *how* to do it, it belongs in an Engine. If it reaches out to *where* data or services live, it belongs in a Resource Accessor. If it supports everything but belongs to no domain, it is a Utility.

6.1 Managers

Managers coordinate operation flow and encapsulate high-level business orchestration. They represent business intent and workflow coordination and should remain stable over time.

Managers **MUST NOT** perform heavy computation. Managers **MUST NOT** directly share state. Managers **MAY** communicate with other managers only through queued, fire-and-forget mechanisms. Managers **MAY** invoke engines and resource accessors directly.

Only Managers emit and consume events. Async dispatch, pub/sub, and queued messaging are Manager-layer concerns. Engines and Resource Accessors operate synchronously within a request.

Illustrative Manager examples:

- *Order Processing Manager* — Coordinates the lifecycle of an order by sequencing steps such as validation, pricing, fulfillment, and confirmation without embedding business rules.
- *Customer Interaction Manager* — Orchestrates user-facing workflows, invoking engines to fulfill intent and translating technical outcomes into business-level results.
- *Batch Execution Manager* — Coordinates scheduled or bulk operations by partitioning work, invoking engines per unit of work, and handling workflow-level retries.

Managers should not implement business rules, perform data aggregation, or contain persistence or integration logic. Their primary responsibility is to express what the system is trying to accomplish, not how it is achieved.

6.2 Engines

Engines execute complex business rules, transformations, or computationally intensive operations. They encapsulate the logic most likely to change due to policy shifts, experimentation, or optimization.

Engines **MUST NOT** communicate with other engines. Engines **MUST NOT** use queued or pub/sub mechanisms. Engines **MAY** call dependent services directly.

Illustrative Engine examples:

- *Pricing Engine* — Calculates prices based on rules, tiers, and promotions, evolving frequently as business strategies change.
- *Eligibility or Policy Engine* — Evaluates compliance, qualification, or constraint logic driven by regulatory or policy updates.
- *Recommendation or Matching Engine* — Performs scoring, ranking, or matching algorithms that change due to tuning or experimentation.

- *Transformation Engine* — Converts, normalizes, or enriches data representations without awareness of workflow or persistence.

Engines must not coordinate workflows, interact with messaging infrastructure, or embed persistence concerns. They are invoked by managers and remain unaware of the broader execution context.

6.3 Resource Accessors

Resource accessors manage interaction with persistence layers, external systems, vendors, and infrastructure-facing resources. They isolate environmental and integration volatility from the rest of the system.

Resource accessors **MUST NOT** communicate with engines or other resource accessors. Resource accessors **MUST NOT** use queued or pub/sub mechanisms. Resource accessors **MAY** call dependent services directly.

Illustrative Resource Accessor examples:

- *Order Repository Accessor* — Encapsulates database access and schema evolution, shielding the system from persistence changes.
- *External Payment Gateway Accessor* — Manages vendor-specific APIs, retries, error translation, and versioning.

- *Messaging or Queue Accessor* — Publishes or consumes messages while hiding transport protocols and infrastructure configuration.
- *Configuration or Secrets Accessor* — Retrieves environment-specific configuration values without leaking deployment concerns.

Resource accessors must not apply business rules, coordinate workflows, or make policy decisions. Their responsibility is to interact with external resources reliably and predictably.

6.4 Utilities

Utilities encapsulate cross-cutting concerns that apply broadly across the system and evolve independently of business workflows. They are orthogonal to core behavior and should remain free of domain-specific knowledge.

Illustrative Utility Component examples:

- *Logging Utility* — Provides standardized logging APIs, log formatting, and correlation identifiers without embedding business meaning.
- *Monitoring and Telemetry Utility* — Collects metrics, traces, and health signals to support observability without influencing execution flow.

- *Error Classification and Mapping Utility* — Normalizes and categorizes errors across components, translating low-level failures into consistent error types.
- *Feature Flag Utility* — Enables conditional behavior toggling and experimentation while remaining decoupled from business rules.
- *Security Utility* — Supports cryptographic operations, token validation helpers, or hashing functions without making authorization decisions.

Utilities must not coordinate workflows, enforce business policy, or directly interact with external systems on behalf of managers or engines. Their role is to provide shared capabilities that reduce duplication while preserving architectural boundaries.

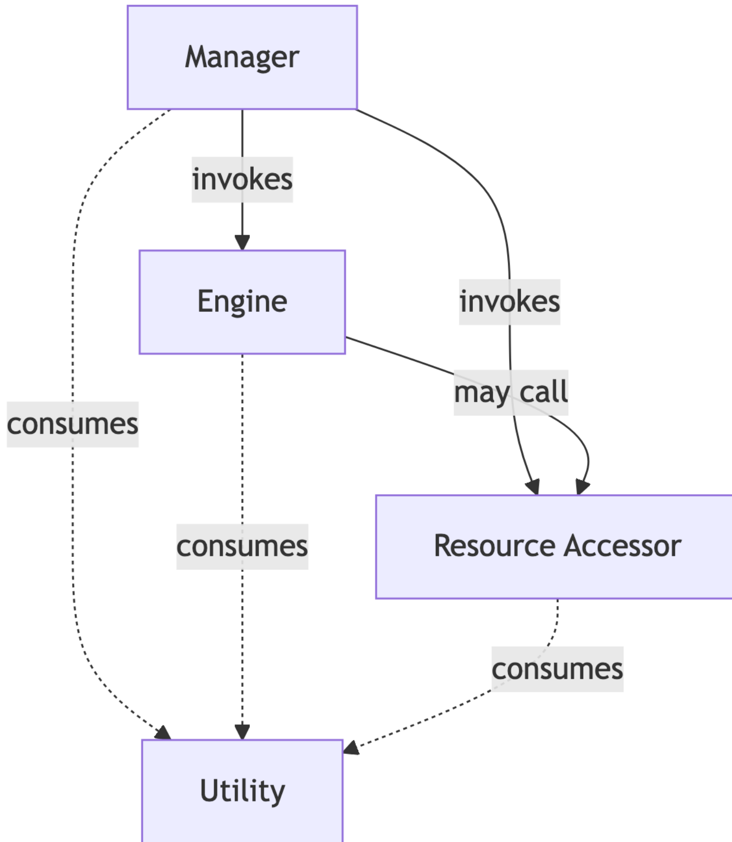


Figure 2 — Component Roles and Communication Rules: illustrates the four component roles in VBD — Managers, Engines, Resource Accessors, and Utilities — along with their permitted communication patterns and constraints.

✦ ✦ ✦

7. Core Use Cases as Architectural Validation

Core use cases are end-to-end scenarios that exercise all relevant ancillary behaviors. These core use cases serve as validation mechanisms, ensuring that architectural boundaries support real execution paths without introducing hidden coupling or responsibility leakage.

If a core use case requires bypassing defined communication rules, the architecture should be reconsidered.

✦ ✦ ✦

8. Continuous Evaluation and Architectural Evolution

Volatility analysis is not a one-time activity. As systems evolve, new volatility axes emerge and existing assumptions may become invalid.

To maintain architectural integrity:

- Monitor changes in requirements, infrastructure, and usage patterns
- Conduct periodic architectural reviews
- Update volatility classifications and component boundaries
- Communicate architectural changes clearly to all stakeholders

9. Architectural Watchpoints

Volatility-Based Decomposition is most effective when applied intentionally and proportionally. Like any architectural approach, it introduces forces that must be actively managed over time. The following watchpoints highlight areas that warrant ongoing attention rather than serving as hard limitations.

Decomposition Granularity Over-decomposition can increase cognitive overhead and coordination cost. Architects should monitor whether component boundaries continue to align with meaningful volatility axes or whether decomposition has become finer than the rate of change justifies.

Organizational Discipline The effectiveness of VBD depends on consistent adherence to component roles and communication rules. When these boundaries are eroded — often in the interest of short-term delivery — volatility begins to leak across components, reintroducing change coupling.

System Scale and Lifespan Smaller or short-lived systems may not benefit from full application of VBD. Architects should assess expected system longevity and change rate to determine the appropriate level of rigor.

Evolution of Volatility Axes Volatility is not static. Axes that were once stable may become volatile as business strategy, technology, or regulatory environments change. Periodic architectural review is required to ensure boundaries remain aligned with current realities.

✦ ✦ ✦

9A. Volatility-Based Decomposition in Real-World Systems

The principles of Volatility-Based Decomposition are most clearly demonstrated when applied to real, long-lived software systems operating under continuous change. The following examples are intentionally domain-agnostic and generalized, focusing on architectural forces rather than implementation specifics.

Example 1: Long-Lived Enterprise Platform

In large enterprise platforms supporting multiple business units, functional requirements evolve unevenly. Core workflows may remain stable for years, while regulatory logic, reporting requirements, and integration points change frequently.

Applying VBD in this context typically reveals:

- Managers remain stable, coordinating high-level workflows and orchestration that change infrequently.
- Engines absorb volatility related to business rules, calculations, policy enforcement, and workflow-specific decisioning.
- Resource Accessors isolate churn driven by database migrations, schema evolution, vendor swaps, and third-party integrations.
- Utilities encapsulate cross-cutting concerns such as auditing and compliance logging, which evolve independently of business logic.

This decomposition localizes regulatory- and integration-driven change, preventing widespread refactoring when external requirements shift.

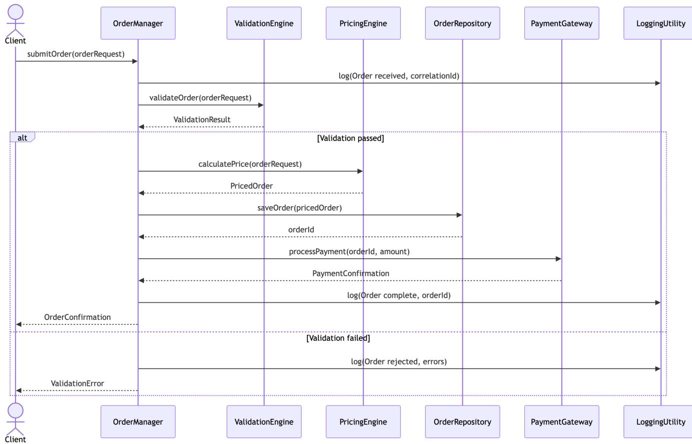


Figure 3 — Core Use Case: Order Processing: demonstrates how a core use case flows through VBD component roles, with Managers coordinating, Engines executing business logic, Accessors handling persistence, and Utilities providing cross-cutting capabilities.

Example 2: Integration-Heavy or Platform Systems

Systems that serve as integration hubs — connecting internal services, external partners, and third-party APIs — experience high environmental and infrastructural volatility. APIs version independently, protocols evolve, and reliability characteristics vary widely.

Under VBD:

- Managers remain insulated from integration complexity, focusing on orchestration and intent.

- Engines handle transformation, enrichment, and aggregation logic without direct exposure to external systems.
- Resource Accessors encapsulate protocol translation, retries, circuit breaking, vendor-specific behavior, and API version management.

This structure allows integrations to be replaced or upgraded with minimal impact beyond the accessor layer.

Example 3: Rapidly Evolving Product Systems

Product-focused systems often face intense functional volatility driven by experimentation, user feedback, and market pressure. Performance and scalability concerns may also shift rapidly as adoption grows.

VBD enables such systems to evolve by:

- Keeping Managers stable, preserving workflow integrity and orchestration even as underlying behavior shifts.
- Allowing Engines to absorb frequent algorithmic, rules, and decision-logic changes.
- Using Resource Accessors to isolate churn in persistence choices, external dependencies, and infrastructure-facing concerns.
- Using Utilities to adapt cross-cutting needs such as observability and feature flagging without contaminating core logic.

This separation enables rapid iteration while maintaining architectural coherence.

Observed Outcomes

Across these scenarios, consistent outcomes emerge:

- Change is localized rather than systemic.
- Architectural intent remains visible and enforceable.
- Teams can evolve different parts of the system independently.

These results reinforce VBD's central premise: aligning architectural boundaries with volatility produces systems that are resilient over time.

✦ ✦ ✦

9B. Practitioner Observations

The following observations emerge from applying Volatility-Based Decomposition across multiple systems over extended periods. They are not specific to any organization or domain. Rather, they describe recurring patterns, tensions, and emergent behaviors that practitioners are likely to encounter when VBD is applied consistently as a system matures.

Observation 1: Volatility Migration

Volatility axes are not permanent. What was stable for years can become intensely volatile due to external forces, and what was once highly volatile can stabilize as a domain matures. A common example is compliance logic: rules that remained unchanged for a decade may suddenly enter a period of rapid, continuous change when new regulation is introduced. Architectures that hardcoded compliance assumptions into Managers or even Utilities find themselves performing invasive surgery across the system. The practical implication is that volatility classification must be revisited periodically. Systems designed under VBD absorb this migration more gracefully because the boundaries already exist — the work becomes reclassification and, at worst, extraction of logic from one role into another, rather than a fundamental restructuring.

Observation 2: Accessor Accumulation

As systems grow, Resource Accessors tend to proliferate. Each new vendor integration, database, external API, or infrastructure dependency typically receives its own Accessor, which is correct according to VBD principles. Over time, however, the sheer number of Accessors can become a management burden. The diagnostic question is whether multiple Accessors share the same volatility profile — if two Accessors

change for the same reasons, at the same rate, and are owned by the same team, consolidation is appropriate. If they change independently, they should remain separate regardless of superficial similarity. The temptation to merge Accessors for tidiness should be resisted when their volatility profiles differ, as doing so reintroduces the coupling VBD was designed to eliminate.

Observation 3: Engine Extraction

Logic that begins inside a Manager has a natural tendency to migrate into Engines over time. In early development, when the business rules are not yet fully understood, teams often embed decision logic directly in the Manager's orchestration flow because the rules seem simple or because the team is still discovering what the rules actually are. As the system matures and the rules become clearer, more complex, or more independently volatile, practitioners consistently find themselves extracting that logic into dedicated Engines. This is not a failure of initial design — it is a healthy and expected progression. VBD accommodates this migration by design: the Manager's orchestration structure remains intact while the extracted Engine absorbs the volatile logic behind a clean interface.

Observation 4: The Utility Trap

Utilities are the most frequently misused component role in VBD. Because they are accessible to all other roles and represent “shared” capabilities, there is a persistent temptation to place logic in a Utility simply because multiple consumers need it. The diagnostic is straightforward: if the Utility contains domain knowledge — if it knows about orders, customers, pricing, or any business concept — it is not a Utility. It is an Engine or an Accessor that has been misclassified for convenience. True Utilities are domain-agnostic: logging, cryptographic hashing, date formatting, correlation identifier generation, metric collection. When a Utility begins accumulating business-aware conditional logic, it should be reclassified and relocated before it becomes a hidden coupling point that undermines the entire decomposition.

Observation 5: Conway’s Law Alignment

VBD interacts productively with Conway’s Law — the observation that system structure tends to mirror organizational communication structure. When component boundaries are aligned with volatility axes, team ownership naturally follows. The team that owns pricing policy owns the Pricing Engine. The team that manages vendor relationships owns the relevant Resource Accessors. This alignment reduces cross-team coordination costs because changes are

localized not only architecturally but organizationally. Conversely, when VBD boundaries conflict with team structure, one of two things must change: either the architecture is adjusted to reflect organizational reality, or the organization is restructured to match the architecture. In practice, the most successful outcomes occur when both are considered together.

Observation 6: Emergent Configuration-Drivenness

Systems built under VBD tend to become configuration-driven over time, even when this was not an explicit design goal. The mechanism is straightforward: because Engines encapsulate business rules behind stable interfaces, and because those rules change frequently, teams naturally begin externalizing rule parameters into configuration rather than modifying code for each change. Pricing tiers become configuration tables. Eligibility thresholds become feature flags. Validation constraints become schema-driven. This emergent behavior is a structural consequence of properly isolating volatile logic — once the volatility boundary is clean, the path of least resistance for absorbing change shifts from code modification to configuration update. Practitioners should recognize and embrace this tendency rather than treating it as accidental.

Observation 7: The Stability Paradox of Managers

Managers are designed to be the most stable component in a VBD system, yet they are often the first component written and the last to be understood correctly. Early in a system's life, Managers tend to accumulate logic that belongs elsewhere — business rules, data transformation, error handling policy — because the team has not yet identified which concerns are independently volatile. The paradox is that achieving Manager stability requires the most architectural discipline, precisely because Managers sit at the top of the invocation hierarchy and are the most convenient place to add logic quickly. Teams that enforce Manager stability from the outset consistently report lower long-term maintenance costs, while teams that allow Managers to accumulate non-orchestration logic find themselves performing costly extractions later.

Observation 8: Boundary Pressure as a Health Signal

In mature VBD systems, the points where teams feel the most pressure to violate communication rules serve as reliable indicators of architectural misalignment. When an Engine needs to call another Engine, it typically signals that the Manager above them is not properly coordinating the workflow, or that the two Engines should be merged because they share a volatility axis. When a Resource Accessor

begins applying business rules, it signals that an Engine is missing from the architecture. Rather than treating boundary violations as failures of discipline alone, experienced practitioners use them as diagnostic signals — the pressure to violate a rule reveals where the current decomposition no longer matches the system's actual volatility profile.

✦ ✦ ✦

10. Conclusion

Volatility-Based Decomposition provides a disciplined approach to software architecture that treats change as an explicit design constraint rather than an afterthought. By identifying functional, non-functional, cross-cutting, and environmental sources of volatility, architects can align system boundaries with the forces most likely to cause erosion over time.

Through established component roles, strict communication rules, and continuous validation against core use cases, Volatility-Based Decomposition supports the creation of architectures that remain adaptable as systems grow in scale, complexity, and organizational impact.

In many organizations, architectural weakness is revealed not by the initial change itself, but by the unintended side effects that follow. A seemingly localized modification triggers a cascade of

downstream impacts, forcing broad testing cycles, emergency fixes, and production instability. These domino effects erode confidence, slow delivery, and increase operational risk.

By aligning architectural boundaries with volatility and validating changes against a small set of core use cases, VBD localizes change and limits its blast radius. While no approach can eliminate change, volatility-based decomposition reduces the likelihood that a single modification will destabilize unrelated parts of the system, preserving architectural integrity and lowering long-term maintenance costs.

As software systems continue to operate in increasingly dynamic environments, architectures that explicitly design for volatility will prove more resilient than those optimized solely for present-day requirements.

✦ ✦ ✦

Appendix A: Glossary

Blast Radius — The extent of system impact caused by a single change. VBD aims to minimize blast radius by aligning boundaries with volatility axes.

Change Coupling — A condition where modifying one component forces changes in another, even when the two have no logical dependency. Indicates misaligned volatility boundaries.

Communication Rules — The explicit constraints governing which component roles may invoke which others, and through what mechanisms. Prevents dependency erosion and preserves volatility isolation.

Component Role — One of the four architectural roles assigned to a component based on the type of concern it encapsulates: Manager, Engine, Resource Accessor, or Utility.

Core Use Case — A high-level system behavior that defines primary business value and exercises all component tiers. Used to validate architectural decisions and communication rules.

Decomposition — The process of breaking a system into components along boundaries that align with anticipated sources of change.

Engine — A component that encapsulates business rules, calculations, transformations, and policy logic. Engines answer *how* the system performs its work and absorb functional volatility as business rules evolve.

Environmental Volatility — Change driven by infrastructure platforms, third-party services, vendor APIs, deployment models, and hosting environments.

Functional Volatility — Change driven by evolving business behavior, workflows, features, regulations, and user requirements.

Information Hiding — Parnas's principle of decomposing systems based on design decisions likely to change. VBD generalizes this from individual modules to architectural boundaries.

Manager — A component that coordinates workflow, sequencing, and intent. Managers answer *what* the system does and remain stable over time by containing no business logic or infrastructure awareness.

Non-Functional Volatility — Change driven by performance, scalability, reliability, and other quality-of-service requirements.

Orchestration — The coordination of workflow steps, sequencing, and intent. Separated from execution in VBD to allow workflows and business rules to evolve independently.

Resource Accessor — A component that isolates interactions with databases, external APIs, vendors, and infrastructure. Resource Accessors answer *where* data and services live and shield the system from environmental volatility.

Stability — The property of a component that changes infrequently relative to others. Managers are designed to be the most stable components in a VBD system.

Utility — A component that encapsulates cross-cutting concerns such as logging, monitoring, security, and observability. Utilities are orthogonal to business logic and may be called by any other component.

Volatility — The likelihood, frequency, and impact of change affecting a system responsibility or requirement. The primary organizing force in VBD.

Volatility Axis — A dimension along which change is expected to occur. The four primary axes are functional, non-functional, cross-cutting, and environmental.

Volatility Boundary — An architectural seam placed to contain a specific source of change, preventing it from propagating to unrelated components.

Volatility Migration — The phenomenon where a concern shifts from one volatility axis to another over time, requiring reassessment of component boundaries.

✦ ✦ ✦

Appendix B: Applicability Checklist

Volatility-Based Decomposition is particularly well-suited for systems that:

- Are expected to evolve over multiple years

- Operate across changing infrastructure or regulatory environments
- Support multiple teams or organizational boundaries
- Require long-term maintainability and extensibility

✦ ✦ ✦

Appendix C: Case Study — Multi-Tenant SaaS Billing Platform

This appendix presents a fictional but architecturally realistic case study applying Volatility-Based Decomposition to a multi-tenant SaaS billing platform. The system is responsible for generating invoices, calculating charges based on usage and subscription tiers, applying externally-imposed adjustments across multiple jurisdictions, processing payments through multiple providers, and notifying tenants of billing activity.

C.1 Volatility Analysis

The first step is identifying what changes and what remains stable.

High Volatility:

- **Pricing rules** — Subscription tiers, volume discounts, promotional offers, and per-unit rates change continuously as business strategy evolves. Pricing data lives behind an accessor because where and how prices are stored is itself a volatility axis.
- **External adjustments** — Tax rates, nexus rules, exemption categories, regulatory fees, and reporting obligations change across jurisdictions independently and unpredictably. External tax services and local databases are isolated behind an accessor.
- **Payment providers** — New providers are added, existing providers change APIs, and regional payment methods must be supported. All provider volatility is isolated behind a single accessor.
- **Invoice storage** — Where and how invoices are persisted — schema, storage technology, archival strategy — is isolated behind an accessor owned by the InvoicingEngine.
- **Notification channels** — How tenants are notified (email, SMS, webhook, in-app) changes based on product decisions and tenant preferences.

Low Volatility:

- **Billing lifecycle** — The fundamental sequence of calculating charges, applying adjustments, generating an invoice, collecting payment, and notifying the tenant has remained stable across billing systems for decades.
- **Audit requirements** — The need to maintain a complete, immutable audit trail of all financial transactions is a permanent architectural requirement.

C.2 Component Identification

Managers:

- **BillingManager** — Orchestrates the end-to-end billing lifecycle: delegates pricing calculation, adjustment application, invoice creation and persistence, payment collection, and tenant notification. Contains no business rules. Expresses intent and sequence only. Fires an async event to NotificationManager after payment is confirmed.
- **NotificationManager** — Orchestrates tenant notification delivery. Receives a billing completion event asynchronously from BillingManager and coordinates the appropriate notification channels. Decoupled from the billing flow — notification failure does not affect billing outcome.

Engines:

- **PricingEngine** — Encapsulates all pricing logic: subscription tier calculations, usage-based metering aggregation, volume discounts, and promotional adjustments. Retrieves current pricing rules via PriceAccessor. Discounts are an aspect of pricing strategy and belong here.
- **AdjustmentEngine** — Encapsulates all externally-imposed modifications to a priced transaction: tax calculations, regulatory fees, payment processing surcharges, and any other obligations not owned by the business's pricing strategy. Tax is the primary case — AdjustmentEngine retrieves rates via TaxAccessor and applies jurisdiction-specific rules — but the Engine's boundary is defined by the volatility axis, not the adjustment type. When a new regulatory fee or cross-border surcharge must be applied, AdjustmentEngine is the only component that changes.
- **InvoicingEngine** — Assembles the invoice from priced and adjusted line items, calculates totals, and persists the completed invoice via InventoryAccessor. Owns both creation and persistence — BillingManager receives only the invoiceId in return.

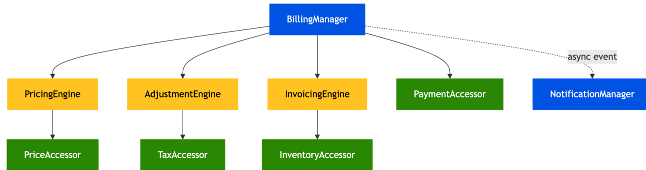
Resource Accessors:

- **PriceAccessor** — Retrieves pricing rules, tier definitions, and discount schedules from the pricing data store. Isolates PricingEngine from changes in pricing data structure, storage technology, and external pricing service APIs.
- **TaxAccessor** — Retrieves tax rates and jurisdiction rules from external tax services (Avalara, TaxJar) or local tax databases. Isolates AdjustmentEngine from changes in data sources and service provider APIs.
- **InventoryAccessor** — Manages persistence of generated invoices, line items, and billing history. Called by InvoicingEngine, not by BillingManager directly.
- **PaymentAccessor** — Manages integration with external payment providers (Stripe, PayPal, bank ACH, and future providers). Translates provider-specific protocols into a uniform PaymentConfirmation interface.

C.3 Component Architecture

Figure C.1 — Billing Platform Component Architecture: BillingManager orchestrates the billing lifecycle, delegating to each Engine and calling PaymentAccessor directly. Each

Engine owns its own accessor boundary. NotificationManager receives an async event after payment is confirmed and operates independently of the billing flow.



diagram

C.4 Communication Rules Applied

Source	Target	Permitted	Rationale
BillingManager	PricingEngine	Yes	Manager invokes Engine
BillingManager	AdjustmentEngine	Yes	Manager invokes Engine
BillingManager	InvoicingEngine	Yes	Manager invokes Engine
BillingManager	PaymentAccessor	Yes	Manager invokes Accessor — after invoice is complete
BillingManager	NotificationManager	Yes (async)	Manager-to-Manager via event — fire and forget
BillingManager	InventoryAccessor	**No**	Invoice persistence is InvoicingEngine's responsibility
PricingEngine	AdjustmentEngine	**No**	Engine-to-Engine communication prohibited
PricingEngine	PriceAccessor	Yes	Engine calls its own Accessor
AdjustmentEngine	TaxAccessor	Yes	Engine calls its own Accessor
InvoicingEngine	InventoryAccessor	Yes	

Source	Target	Permitted	Rationale
			Engine calls Accessor to complete its responsibility
PaymentAccessor	InventoryAccessor	**No**	Accessor-to-Accessor communication prohibited

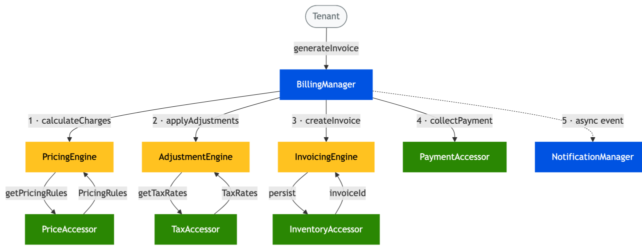
BillingManager does not call InventoryAccessor directly. Invoice persistence is encapsulated within InvoicingEngine, which calls InventoryAccessor as part of fulfilling its own responsibility. BillingManager receives only the invoiceId in return.

C.5 Core Use Case Walkthrough: Generate and Process Monthly Invoice

1. **BillingManager** receives a trigger to generate the monthly invoice for a tenant.
2. **BillingManager** calls **PricingEngine** with the tenant's usage data and subscription tier. PricingEngine retrieves current pricing rules via **PriceAccessor**, calculates line items, applies discounts and pro-rations, and returns a priced line item set.

3. **BillingManager** calls **AdjustmentEngine** with the priced line items and transaction context (jurisdiction, payment method, tenant classification). AdjustmentEngine applies all externally-imposed obligations — retrieving tax rates via **TaxAccessor**, applying jurisdiction-specific rules, and layering in any applicable fees or surcharges — and returns the adjusted line items. BillingManager has no knowledge of which adjustment types were applied or how many.
4. **BillingManager** calls **InvoicingEngine** with the priced and adjusted line items. InvoicingEngine assembles the invoice document, calculates totals, and persists the completed invoice via **InventoryAccessor**. BillingManager receives only the invoiceId in return.
5. **BillingManager** calls **PaymentAccessor** with the invoiceId, total amount, and payment method. PaymentAccessor interacts with the configured payment provider, handles provider-specific protocols, and returns a PaymentConfirmation.
6. **BillingManager** fires an async event to **NotificationManager** with the billing result. NotificationManager operates independently — it coordinates notification delivery without blocking or affecting the billing outcome.

Figure C.2 — Core Use Case: Generate and Process Monthly Invoice. Steps 1–4 are sequential — pricing before adjustments, adjustments before invoice, invoice before payment. Step 5 is fire-and-forget.



diagram

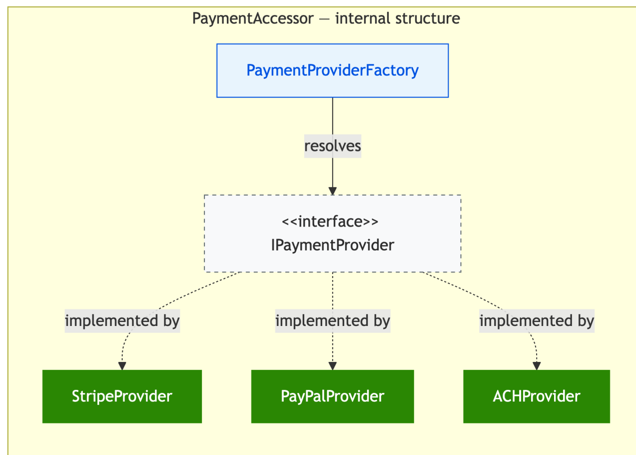
C.6 Internal Design: PaymentAccessor and the Strategy Pattern

VBD defines the boundaries between components and the rules governing communication across those boundaries. It says nothing about how a component is organized internally. That is deliberate — once a boundary is established, the team that owns the component is free to structure its internals however best serves maintainability, testability, and extensibility, without any obligation to the rest of the system.

PaymentAccessor illustrates this clearly. Its contract to the outside world is simple: receive a payment request, return a PaymentConfirmation or

failure. How it fulfills that contract is entirely its own concern. Internally, PaymentAccessor implements the Strategy pattern — a **PaymentProviderFactory** resolves the correct **IPaymentProvider** implementation based on the context of the call (payment method, region, tenant configuration, or any other dispatch criterion), and delegates execution to it.

Figure C.3 — PaymentAccessor internal structure. The Strategy pattern isolates provider-specific implementations behind IPaymentProvider. PaymentProviderFactory resolves the correct implementation at runtime based on call context. The rest of the system sees none of this.



diagram

Because BillingManager holds only a reference to PaymentAccessor — not to any of its internals — the team owning PaymentAccessor can:

- **Add a new provider** by writing a new IPaymentProvider implementation. No other component changes.
- **Refactor the factory** — change dispatch logic, add caching, introduce weighted routing — without touching BillingManager or any other component.
- **Replace the internal architecture entirely** — swap the Strategy pattern for a plugin registry, a configuration-driven resolver, or anything else — as long as the external contract is preserved.
- **Test provider implementations in isolation**, mock at the IPaymentProvider boundary, and evolve internal test strategy independently of the rest of the system.

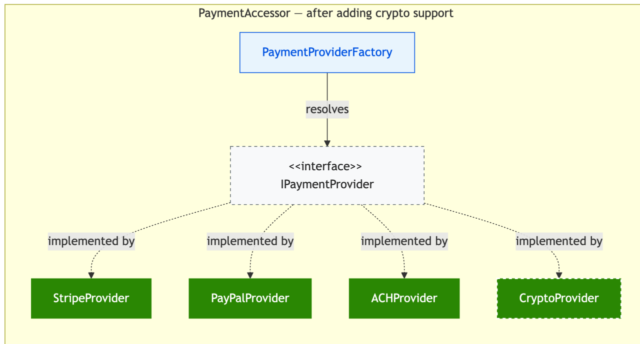
This is the compounding benefit of a well-placed boundary. VBD does not prescribe how PaymentAccessor is built — it only prescribes where its boundary is and what may cross it. Everything inside that boundary is a local decision, free from the coordination cost that would accompany any change visible to the rest of the system.

The same principle applies to every component. PricingEngine may internally maintain a chain of discount calculators. InvoicingEngine may use a template engine or a document builder. AdjustmentEngine may dispatch to multiple adjustment strategies in sequence. These are internal implementation choices. The system does not know and does not need to know.

C.7 Volatility Isolation Demonstrated: Adding Crypto Payments

The business decides to accept cryptocurrency payments. A new **CryptoProvider** implementation is written — handling blockchain confirmation logic, wallet address validation, exchange rate locking, and the crypto processor's API — and registered with the factory. That is the entirety of the change.

Figure C.4 — CryptoProvider added to PaymentAccessor. One new class, registered with the factory. The interface contract is unchanged. No other component in the system is modified or redeployed.



diagram

BillingManager still calls `collectPayment(invoiceId, amount, paymentMethod)` — the same call it has always made. It has no knowledge that a new payment type exists. PricingEngine, AdjustmentEngine, InvoicingEngine, and NotificationManager are untouched. The factory now routes crypto payment requests to CryptoProvider; everything else routes exactly as before.

One class added. Eight components untouched. The payment provider axis of volatility is fully contained within PaymentAccessor, and the boundary enforces that containment regardless of how many providers are added over time.

✦ ✦ ✦

References and Influences

The concepts presented in this paper are informed by, and build upon, established work in software architecture, component design, and object-oriented systems. Volatility-Based Decomposition is not presented as a novel invention, but as a practitioner-oriented articulation of principles that have emerged and matured through decades of architectural thought and practice.

Juval Löwy Löwy, Juval. *Righting Software.* Addison-Wesley, 2019. Löwy, Juval. *Programming .NET Components.* O'Reilly Media, 2005.

Juval Löwy's work is the primary foundation for Volatility-Based Decomposition. His IDesign methodology explicitly frames volatility as the dominant architectural force and emphasizes designing systems around anticipated change rather than static functionality. The Manager, Engine, and Resource Accessor role taxonomy, along with the associated communication rules and component interaction discipline described in this paper, are derived directly from IDesign training. This paper builds upon Löwy's work by consolidating these principles into a single, cohesive decomposition process and demonstrating their application across diverse system contexts.

David L. Parnas Parnas, David L. "On the Criteria To Be Used in Decomposing Systems into Modules." *Communications of the ACM*, 1972.

Parnas introduced the principle of information hiding, arguing that systems should be decomposed based on the design decisions most likely to change. This idea is foundational to Volatility-Based Decomposition. VBD can be viewed as a system-level extension of Parnas's insight, generalizing information hiding beyond individual modules to entire architectural boundaries aligned with volatility axes.

Gang of Four (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides) Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

The Gang of Four cataloged recurring object-oriented patterns that encapsulate and localize variation at the class and collaboration level. These patterns demonstrate how volatility can be managed through indirection, composition, and role separation. Volatility-Based Decomposition extends this pattern-based thinking from the object scale to the architectural scale, applying the same principles of variation isolation across components and services.

Robert C. Martin Martin, Robert C. *Clean Architecture.* Pearson, 2017. Martin, Robert C. *Agile Software Development: Principles, Patterns, and Practices.* Pearson, 2002.

Martin's work emphasizes responsibility-driven design, stable dependency direction, and the separation of policy from implementation details. These ideas complement VBD's focus on isolating volatile concerns and enforcing communication rules that prevent dependency inversion from eroding architectural boundaries over time.

Eric Evans Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley, 2003.

Domain-Driven Design introduces bounded contexts as strategic boundaries for managing conceptual and organizational complexity. While VBD does not prescribe domain boundaries, it aligns with Evans's emphasis on explicit boundary definition. Volatility-Based Decomposition can coexist with DDD by treating bounded contexts as one possible axis of volatility among others, such as regulatory change or infrastructure churn.

Gregor Hohpe and Bobby Woolf Hohpe, Gregor; Woolf, Bobby. *Enterprise Integration Patterns.* Addison-Wesley, 2003.

Hohpe and Woolf formalized integration and messaging patterns that address the volatility inherent in distributed systems. Their work informs the disciplined communication rules emphasized in VBD, particularly around asynchronous messaging, decoupling, and the containment of integration complexity within well-defined architectural boundaries.

Mary Shaw and David Garlan Shaw, Mary; Garlan, David. *Software Architecture: Perspectives on an Emerging Discipline.* Pearson, 1996.

Shaw and Garlan helped establish software architecture as a first-class discipline distinct from programming and design. Their work provides the conceptual grounding for system-level decomposition approaches like Volatility-Based Decomposition, reinforcing the idea that architectural structure must be reasoned about explicitly and evaluated continuously as systems evolve.

✦ ✦ ✦

Author’s Note

Volatility-Based Decomposition (VBD), including its terminology, volatility-first orientation, component role taxonomy, and communication rules, originates from Juval Löwy’s IDesign methodology. This paper does not introduce a new architectural approach. It

provides a consolidated, practitioner-oriented articulation of VBD, emphasizing decomposition mechanics, validation strategies, and real-world application across long-lived software systems.

The intent of this paper is to serve as a durable reference that translates volatility-centered principles into a form suitable for consistent application, discussion, and review within modern engineering organizations.

✦ ✦ ✦

Distribution Note

This document is provided for informational and educational purposes. It may be shared internally within organizations, used as a reference in architectural discussions, or adapted for non-commercial educational use with appropriate attribution. This paper does not represent official policy, standards, or architectural mandates of any current or former employer. All examples are generalized and abstracted to avoid disclosure of proprietary or sensitive information.

APPENDIX C

Experience-Based Decomposition

Experience-Based Decomposition (EBD)

Organizing Interface Architecture Around Human Intent: A Framework

Author: William Christopher Anderson **Date:** April 2026 **Version:** 1.0

✦ ✦ ✦

Executive Summary

The most common complaint about user interfaces in long-lived products is not that individual screens are poorly designed. It is that changing anything requires touching everything. A wizard gains a conditional step; navigation logic scattered across twelve

components must be updated. A new user segment arrives; flow assumptions baked into leaf components need untangling. A new locale is added; string literals buried in state machines surface one by one.

The structural cause of this is well understood in backend systems — responsibilities distributed across the wrong boundaries — yet the interface layer repeatedly inherits the same problem, dressed differently. Screens are not boundaries. Components are not boundaries. Routes are not boundaries. None of these are units of human purpose, which is the only boundary that remains stable as products evolve.

Experience-Based Decomposition addresses this by establishing human intent as the organizing principle of interface structure. It defines three architectural tiers:

- **Experiences** represent complete user journeys — composable, stable, and bounded by purpose rather than screen count.
- **Flows** encapsulate the goal-directed sequences within an experience. They own one goal, one set of accumulated state, and one set of exit conditions.
- **Interactions** are atomic. A selection, an input, a confirmation. They cannot be meaningfully subdivided from the user's perspective.

Shared utility components handle cross-cutting concerns — locale, theme, validation, alerts — without acquiring knowledge of any particular user purpose. A small set of core user journeys validates that the architecture supports real human intent without boundary leakage.

When this structure is in place and backed by an event-driven communication model, something more powerful becomes possible: experiences can be defined and composed through configuration rather than code. An organizational configuration, a feature flag, an audience-specific ruleset — any of these can alter what an experience contains, in what order flows execute, and how the backend responds to their completion events. The interface becomes programmable at the intent level, not just the component level.

EBD applies most directly to long-lived products, systems serving multiple audience types with distinct mental models, and any product that must remain coherent across years of continuous evolution.

✦ ✦ ✦

Abstract

Product interfaces operate under continuous revision. Requirements arrive from user research, competitive pressure, regulation, and organizational restructuring,

each demanding changes that rarely respect the edges of existing screens or components. Conventional approaches to interface construction organize code around what the system presents rather than what the user intends to accomplish. As the product grows, every meaningful change touches more than it should. The cost of evolution eventually outpaces the value it delivers.

Experience-Based Decomposition is a decomposition framework that treats human intent as the primary structural force in UX design. It introduces a three-tier hierarchy — Experiences, Flows, and Interactions — defines explicit roles, responsibilities, and communication rules for each, and validates structural decisions against a small set of core user journeys. It further describes how an event-driven communication model, when combined with external configuration, enables experiences to be composed and routed dynamically — decoupling UX structure from code in the same way that Volatility-Based Decomposition decouples system behavior from system structure. This paper provides a practitioner-oriented articulation of EBD, covering its structural model, communication discipline, configuration-driven composition, and application across real product contexts.

✦ ✦ ✦

1. Introduction

Interface design has accumulated considerable practice rigor over the past three decades. Interaction design, information architecture, accessibility standards, and component design systems have each contributed real improvements to how products are conceived, constructed, and maintained. Despite this, the underlying architectural problem — how to structure the interface layer so that it absorbs change gracefully — remains largely unsolved in most codebases.

The most expensive interface changes are not redesigns. They are the routine ones: a new field in a form that ripples through five components, a conditional step that requires refactoring a navigation model, a new organizational unit that invalidates assumptions embedded in a dozen places. These changes are expensive not because they are complex in themselves, but because the structure of the codebase offers no natural home for the logic they require. Each modification finds space where it can, distributing responsibility in ways that make the next modification more expensive.

The root cause is misalignment between what the code is organized around and what the user is organized around. Code is organized around screens. Users are organized around purposes.

A user opening a developer onboarding tool is not navigating to a screen. They are pursuing a goal — getting their environment configured correctly — and that goal may involve seven screens, four API calls, three conditional branches depending on their organization's configuration, and a different set of form fields than their colleague sitting next to them. None of that variation lives in the goal itself. The goal is stable. The implementation of it is what varies.

Experience-Based Decomposition organizes the interface layer around that distinction. Structure follows intent. What varies is isolated from what endures. The principles are the same ones that underpin Volatility-Based Decomposition at the system level — applied here to the surface that users actually touch.

✦ ✦ ✦

2. Background

2.1 Screen-Centric Organization and Its Consequences

Frontend frameworks present an implicit model: routes map to pages, pages contain components, components contain logic. This model works well enough when an interface is small and stable. It begins

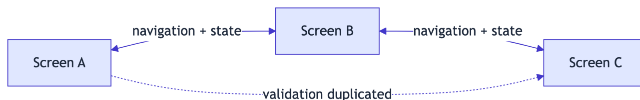
to break down when the product matures and the implementation becomes load-bearing in unexpected places.

The specific failure mode is familiar. State that belongs to a multi-step sequence ends up in a parent component managing all steps. Navigation decisions that depend on user context end up duplicated across components that each independently try to determine what the user should see next. Business logic — what counts as a valid selection, when a step can be skipped — migrates into presentation components because there is no structural location for it that is clearly correct.

Once this pattern takes hold, it is self-reinforcing. New code follows existing patterns. The next engineer to add a feature adds it where similar features already live. The structural debt compounds.

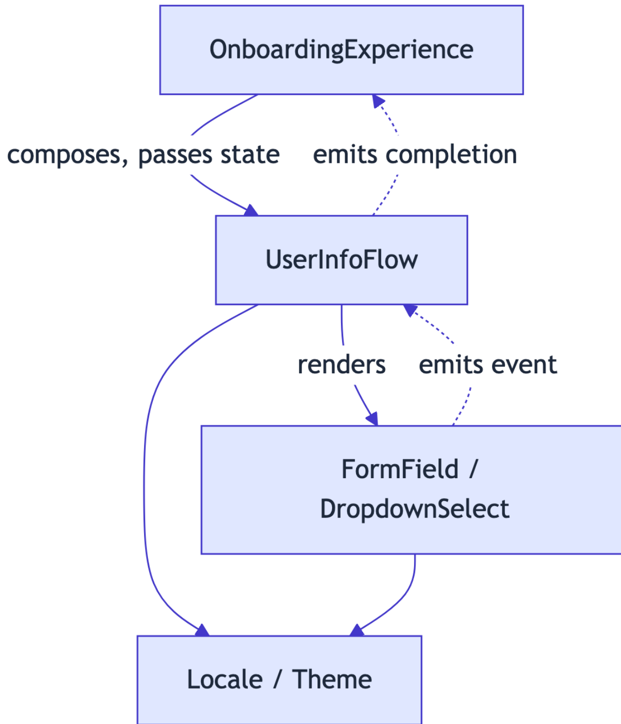
Figure 1a shows the screen-centric failure mode: logic accumulates everywhere because screens are not boundaries. Figure 1b shows EBD's intent-centric structure: each tier has one responsibility.

Figure 1a — Screen-Centric Organization



diagram

Figure 1b — Intent-Centric Organization (EBD)



diagram

2.2 Component-Based Thinking and Its Limits

The dominant response to screen-level fragility has been the component library: shared, composable UI primitives that enforce visual consistency and reduce repetition. This approach has genuine value. Reusable components reduce duplication. Design systems create coherence. Neither addresses decomposition.

A component library answers the question: *how do we build consistent interface elements?* It does not answer the question: *where does navigation logic live? Who owns multi-step state? What happens when a step should be skipped?* Those questions require a structural model, not a component model.

Brad Frost's Atomic Design methodology made an important contribution to this problem by establishing a vocabulary for hierarchical component composition — atoms, molecules, organisms, templates, pages. The hierarchy is real and useful at the component level. What it does not provide is a model for the *behavioral* hierarchy: the distinction between an atomic interaction (selecting an item), a goal-directed sequence (choosing a configuration profile), and a complete user journey (onboarding a developer). Atoms and molecules are structural units. Interactions, flows, and experiences are intentional units. Both taxonomies are necessary; neither replaces the other.

2.3 Goal-Directed Design as a Precursor

Alan Cooper's goal-directed design methodology, articulated in *About Face*, established the foundational insight that interfaces should serve user goals rather than reflect system capabilities. Cooper argued that personas and scenarios should drive interaction

design decisions — that the question is not what the system can do but what the user is trying to accomplish.

EBD extends this insight structurally. It is not sufficient to design around user goals at the specification level if the implementation is organized around screens. The goal-directed principle must persist into the codebase structure, or it dissipates at the moment implementation begins.

✦ ✦ ✦

3. The Four Axes of Interface Volatility

Before defining the structural model, it is useful to characterize where change actually occurs in interface systems — the same volatility analysis that underpins Volatility-Based Decomposition at the system level, applied here to the interface layer.

3.1 Functional Volatility

Changes in what the user is asked to do — new fields, new validation rules, new options, added or removed steps. This is the most frequent category of interface change, driven by product decisions, user research findings, regulatory requirements, and organizational configuration differences. When functional volatility is

not isolated, a single new field type forces changes across presentation, validation, state management, and API layer simultaneously.

3.2 Non-Functional Volatility

Changes in how journeys compose — a new audience type requiring a different path through the same flows, a new entry point, a conditional sub-journey that applies only to a segment of users. Non-functional volatility at the interface layer is less frequent but carries higher impact, because it affects the composition of the interface rather than any individual component.

3.3 Cross-Cutting Volatility

Changes to concerns that propagate broadly without belonging to any particular flow: locale strings, visual themes, form validation conventions, error presentation formats, accessibility requirements. These change independently of user goals and should be isolated in shared utilities that flows and interactions consume without embedding.

3.4 Environmental Volatility

Changes to the external systems the interface depends on — backend API shapes, authentication mechanisms, configuration schemas. This is the interface's equivalent of the integration volatility that

Resource Accessors handle in VBD. It should be contained at a thin boundary layer rather than distributed across flows.

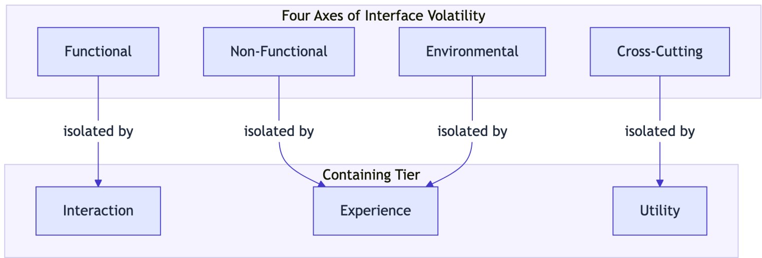


Figure 2 maps each volatility axis to the structural tier responsible for containing it.

✦ ✦ ✦

4. The Structural Model

Experience-Based Decomposition organizes the interface into four structural roles. Each has a defined scope, a bounded set of responsibilities, and explicit rules governing communication with the other roles.

4.1 Experiences

An experience is a complete user journey bounded by a human intent — the arc from initiation to fulfillment. It is the answer to the question: *what did this user come here to accomplish?*

Experiences are the most durable tier. The intent behind a developer onboarding experience — understand the organization's configuration requirements, provide personal context, complete the setup — endures across product iterations, new configuration types, new field types, and new platform targets. An experience should be reconsidered only when the underlying human purpose changes, which is rare.

An experience owns the composition of flows that constitute the journey, the complete state accumulated across all of them, the transitions and terminal conditions, and the top-level progress representation. It is also the only tier that communicates with the backend. When the journey reaches a point where accumulated state should be acted on — a checkpoint, a completion — the experience emits that state as a single coherent event. It holds everything. It decides what comes next.

This gateway role is not incidental. Because the experience has the full picture across all flows, it is the only component that can make a meaningful decision about when and what to send to the backend. A flow knows only its own scope. The experience knows the journey.

An experience **MUST NOT** implement flow-level progression logic. An experience **MUST NOT** render interaction components directly. An experience **MAY**

compose other experiences as bounded sub-journeys. An experience **MAY** receive configuration that determines which flows it contains and in what order. An experience **MAY** call the backend API, passing accumulated journey state as a completion event.

4.2 Flows

A flow is a goal-directed sequence of interactions that accomplishes one discrete outcome within an experience. It is bounded by its goal. One flow, one purpose. The selection of a configuration profile is a flow. The collection of user information is a flow. The confirmation of an installation is a flow. None of these is a screen, though each may render through one.

Flows sit at the most structurally important tier — the one most commonly absent from interface codebases. Because they have no natural representation in screen-first frameworks, flow logic ends up distributed: navigation conditionals in parent components, validation in leaf components, accumulated state in global stores. When the flow has no home, everything becomes its home.

A flow owns its progression through interactions and the state it accumulates across them. It does not communicate with the backend — that is the experience's concern. When a flow finishes, it emits its accumulated state upward. What happens with that state is not the flow's decision.

A flow **MUST NOT** coordinate directly with sibling flows. A flow **MUST NOT** make direct API calls. A flow **MUST NOT** render utility components as owned children — it invokes them. A flow **MAY** emit a completion event carrying accumulated state upward to the experience. A flow **MAY** skip itself entirely if its entry conditions are not met, signaling the experience to advance.

4.3 Interactions

An interaction is the smallest observable unit of user action — atomic and indivisible from the user's perspective. Selecting from a list, entering text, toggling a checkbox, clicking a button, reading a status message. Interactions are the surface. They are what users actually touch.

Interactions are the most volatile tier. The specific fields a given configuration requires changes between organizations. Labels, placeholders, and help text change with locale. Control types — a dropdown becoming a searchable select, a checkbox becoming a toggle — change with design decisions. Because interactions are the most likely to change, they must be the most isolated.

An interaction owns its visual presentation, its internal display state (focused, disabled, errored), and the event it emits when the user acts.

An interaction **MUST NOT** contain flow progression logic. An interaction **MUST NOT** make direct API calls. An interaction **MUST NOT** be aware of adjacent interactions. An interaction **MAY** use utility components for locale rendering, validation display, and alert presentation.

4.4 Utilities

Utilities encapsulate capabilities that cut across all three tiers without belonging to any of them. They are consumed; they do not consume. They have no knowledge of user journeys, specific goals, or the context in which they are invoked.

Locale rendering, visual theme application, progress indication, field validation, and alert presentation are all utility concerns. A locale renderer that knows it is rendering an onboarding string has violated its boundary. The same utility that renders “Next” in English should render “Suivant” in French without any awareness of which flow requested the translation.

A utility **MUST NOT** contain flow progression logic. A utility **MUST NOT** hold domain-specific state. A utility **MUST NOT** coordinate workflows or make decisions on behalf of flows.

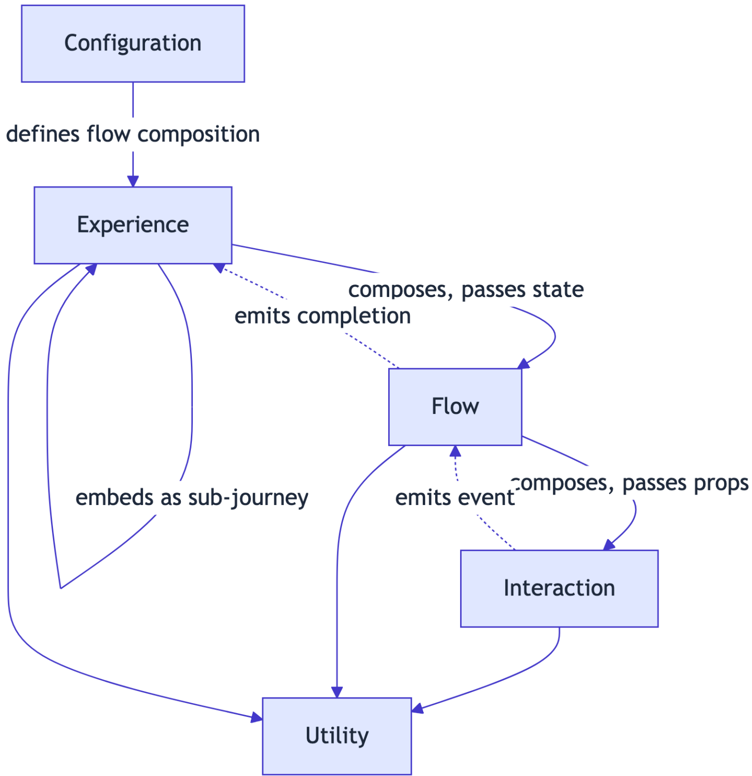


Figure 3 shows the complete structural model and the communication rules between roles.

✦ ✦ ✦

5. Communication Rules

The structural boundaries described above are only as strong as the discipline applied to communication between roles. A flow that calls another flow directly

has already coupled itself to that flow's internal state model, regardless of how cleanly each is implemented in isolation.

The governing rule is directional: **state flows downward; results propagate upward as events.** An experience passes configuration and shared state to flows. Flows pass props and callbacks to interactions. Interactions emit events — they do not call flows. Flows emit completion events — they do not call experiences. No tier reaches sideways. No tier reaches up.

Backend communication is the exclusive domain of the experience. A flow that discovers it needs to call an API has exceeded its scope — it has taken on a decision that belongs one tier up. The experience holds the complete state of the journey. It is the only component with enough information to know when that state is ready to be sent somewhere, what form it should take, and what should happen next in response. Flows surface their state upward and wait for the experience to advance them.

This containment prevents the most common failure mode: a flow that accumulates awareness of sibling flows and backend responses, gradually becoming a de facto experience manager while remaining architecturally classified as a flow.

The second governing rule addresses cross-tier calls: **nothing calls a utility; everything consumes it.** Utilities are invoked inline, not coordinated. A flow does not ask a locale utility to translate a string and then wait for a response. It calls `t("buttons.next")` and receives a string. The distinction matters because a utility that receives callbacks or coordinates timing has begun behaving like a flow.

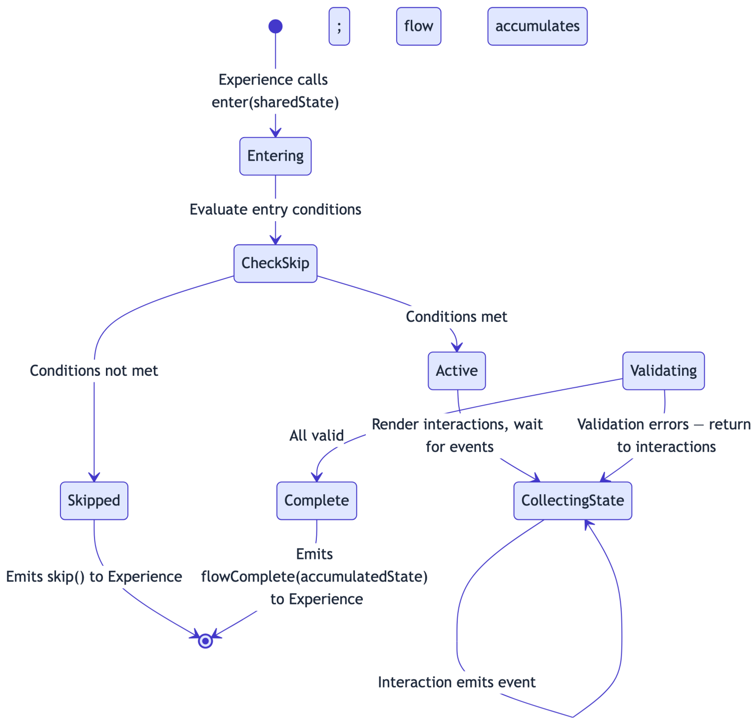


Figure 4 shows the lifecycle of a single flow — how it receives context from the experience, collects state through interactions, and resolves to a completion event.

6. Configuration-Driven Experience Composition

The structural model described in the preceding sections is valuable in its own right. It localizes change, clarifies ownership, and produces codebases where the user's intent is legible in the architecture. When combined with an event-driven communication model and external configuration, it enables something more significant: **experience composition as a runtime concern rather than a compile-time one.**

This is the natural extension of getting the structure right. If an experience is defined as a composition of flows, and that composition is expressed in data rather than code, then the composition itself becomes configurable. An organizational configuration artifact can specify which flows an experience contains, in what order they execute, and what conditions cause any of them to be skipped. Two users with the same interface binary can experience meaningfully different journeys based solely on configuration.

In practice, this requires three things:

A configuration schema that describes experience composition. A schema might specify required flows, optional flows, their execution order, and skip conditions for each. An organization that requires an

additional compliance step presents a flow the standard user never sees. An audience segment with a simplified onboarding path skips flows irrelevant to their context. The same interface binary serves both, driven entirely by configuration.

An event-based communication model between the interface and the backend. When a flow emits a completion event, that event carries the accumulated state from the flow — the selections made, the information provided. A properly structured backend, upon receiving this event, can route the remainder of the transaction differently based on the same configuration that governed the interface. A user who selected the enterprise segment in an OrgProfileFlow may have their provisioning routed through a compliance-validation backend path that the standard user never encounters. The interface does not know about this routing. It emitted an event with a payload. The backend decided what to do.

Backend architecture capable of absorbing this routing. The techniques for achieving this on the backend — separating orchestration from execution, isolating integration concerns, routing workflow decisions through configurable managers — are described fully in *Volatility-Based Decomposition in Software Architecture* (Anderson, 2026). The interface-side pattern and the backend-side pattern are designed to compose: the same organizational

configuration that drives flow selection in the experience layer drives route selection in the manager layer.

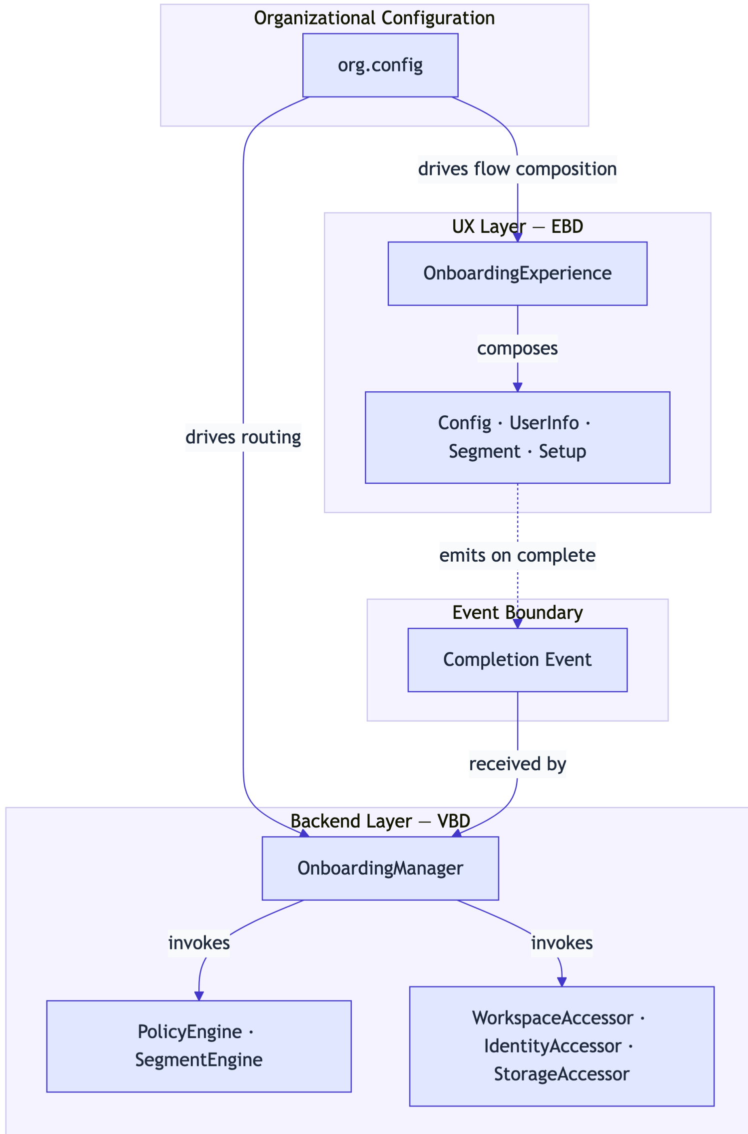


Figure 5 illustrates this end-to-end configuration-driven model.

The consequence of this architecture is worth stating plainly. Once the interface is composed from configuration rather than code, the surface presented to different user segments, organizational units, or deployment contexts can diverge without any code change. A new organizational division that requires an additional configuration flow is accommodated by updating configuration, not by shipping new interface code. Feature flags that progressively surface new experiences to pilot users become a matter of configuration management rather than a deployment. The interface becomes programmable at the level of human purpose.

This is the intended endpoint of applying volatility-first thinking to the interface layer. The structural model localizes change. Configuration-driven composition removes the need for most change to touch code at all.

✦ ✦ ✦

7. Core User Journeys as Architectural Validation

The concept of core use cases as architectural validation mechanisms appears in Volatility-Based Decomposition as a check against boundary leakage. The same technique applies here.

Core user journeys are intentionally narrow. Even complex products have a small number of genuinely core journeys — the high-level human purposes that define what the product is for. Most documented use cases and user stories are variations, edge cases, and exception paths around a much smaller set of essential behaviors.

An interface architecture is structurally sound if each core user journey can be traced cleanly through the Experience → Flow → Interaction hierarchy without bypassing structural rules. Specifically:

- A change within one flow must not require changes in a sibling flow.
- Adding a new interaction to a flow must not require modifying the experience.
- Introducing a new locale string must not require modifying any flow or experience.
- A new organizational configuration variant must produce a different experience composition without code changes.

If any of these conditions fails, the failure points directly to where the architectural boundary has been violated.

A product serving multiple audience types might define three core user journeys — initial onboarding, profile configuration, and workspace setup — each exercising the architecture across different volatility

profiles. The onboarding journey exposes functional and non-functional volatility, as conditional paths and field sets differ by organizational segment. The configuration journey exercises the Experience's environmental boundary directly, translating accumulated user state into a backend API call. The setup journey validates the configuration-driven model, confirming that what the user sees is derived from recorded configuration rather than hardcoded assumptions about their context.

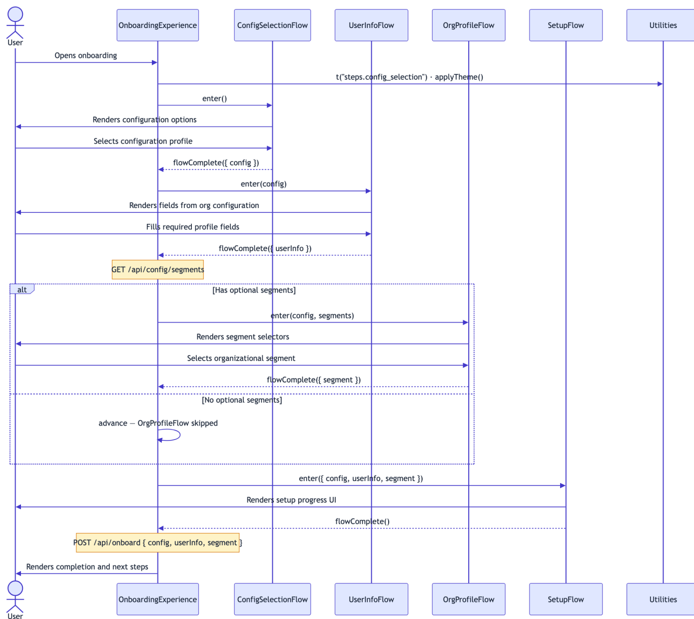


Figure 6 traces a multi-segment onboarding journey through the structural hierarchy.

8. Architectural Watchpoints

Interaction Scope Creep

Interaction components that begin making API calls or managing flow state have absorbed responsibility that belongs one tier up. This pattern typically emerges when a team adds a feature to the most convenient location rather than the correct one. The correction is to extract the logic into the flow and reduce the interaction to its atomic role.

Flow Awareness of Siblings

A flow that checks the state of a sibling flow — even through a shared store — has created a hidden coupling that the structural model was designed to prevent. When a flow needs data produced by another flow, that data should be passed downward from the experience as shared state. If this happens repeatedly, it suggests the experience boundary needs reexamination.

Configuration Complexity Threshold

Configuration-driven composition is powerful, but configuration schemas that grow unconstrained become their own maintenance problem. The same volatility analysis applied to code should be applied to configuration: stable aspects of experience

composition belong in code; variable aspects belong in configuration. The boundary between them should be explicit and documented.

Utility Drift

Utilities that accumulate knowledge of specific flows, user types, or domain rules have crossed from utility into engine territory. A validation utility that behaves differently for enterprise users is no longer a utility. The correction is to move the domain-specific logic into the flow and preserve the utility's domain-agnostic contract.

Experience Depth

Experiences may compose other experiences, but nesting beyond two levels increases coordination cost significantly. Deep experience composition is often a signal that the containing experience has grown into a product rather than a journey, and that the product-level structure needs a separate organizational model.

✦ ✦ ✦

9. Relationship to Volatility-Based Decomposition

EBD and VBD share a single foundational premise: boundaries aligned with change produce systems that remain coherent as they grow. Applied at the system

level, this insight produces the Manager-Engine-Accessor-Utility taxonomy. Applied at the UX level, it produces the Experience-Flow-Interaction-Utility taxonomy. The structural parallels are deliberate.

VBD	EBD
Manager	Experience
Engine	Flow
Resource Accessor	Interaction
Utility	Utility
Core Use Case	Core User Journey
Communication rules (MUST/MUST NOT)	Communication rules (MUST/MUST NOT)
Volatility axis	Interface volatility axis
Configuration-driven routing	Configuration-driven experience composition

The frameworks are designed to compose across the system boundary. When a flow emits a completion event, a VBD manager receives it and routes the subsequent work. Both are responding to the same organizational configuration. Neither knows about the other's internal structure. The contract is the event payload and the configuration schema — nothing more.

This alignment is not merely aesthetic. When both frameworks are applied consistently, changes to organizational configuration propagate through the full stack in a coordinated way: the interface renders different flows, the backend executes different paths, and neither layer required code modification. The configuration is the product. The code is infrastructure.

✦ ✦ ✦

10. Practitioner Observations

The following observations emerge from applying Experience-Based Decomposition across multiple interface systems of varying scale, domain, and maturity. They are not specific to any single product or organization. They describe recurring structural patterns — some beneficial, some pathological — that practitioners encounter as EBD moves from a design-time concept to a living codebase.

10.1 The Flow Proliferation Pattern

Teams new to EBD consistently produce too many Flows in their initial decomposition. The root cause is a confusion between UI screens and behavioral steps. A wizard with five screens does not necessarily contain five Flows. A Flow owns a goal, not a screen. When a user selects a configuration preset and that

selection has no independent lifecycle — no intermediate save, no resumability, no standalone value — it is likely an Interaction within a broader Flow, not a Flow in its own right. The correction is to ask: does this step have its own completion criteria and accumulated state? If not, it belongs inside a Flow that does. Teams that internalize this distinction typically reduce their initial Flow count by 30–50% during their first refactoring pass.

10.2 The Interaction Purity Challenge

Interactions are defined as atomic — the smallest unit of user action that cannot be meaningfully subdivided. In practice, Interactions gradually absorb Flow-level responsibilities. A date-picker Interaction begins making API calls to validate availability. A text-input Interaction starts tracking state across multiple steps. A toggle Interaction begins conditionally showing or hiding sibling Interactions based on business rules. The diagnostic is straightforward: if an Interaction imports services, manages state that survives its own unmounting, or references other Interactions by name, it has absorbed Flow logic. The fix is extraction — move the orchestration concern upward into the owning Flow and reduce the Interaction back to its atomic contract: accept props, render display state, emit events.

10.3 The Experience Scope Question

A recurring architectural question is whether a journey is large enough to be its own Experience or whether it should remain a Flow within a larger Experience. The diagnostic criterion is lifecycle independence. If the journey can be entered, suspended, resumed, and completed without reference to any enclosing journey, it has its own lifecycle and warrants Experience status. If it only makes sense as a step within a broader sequence — if suspending it means abandoning the enclosing journey — it is a Flow. A password-reset journey within a login Experience illustrates the boundary: password reset has its own entry point (a link), its own completion criteria (password changed), and can be abandoned without invalidating the login Experience. It is an Experience, not a Flow. A "confirm your selections" step within an onboarding journey has none of these properties. It is a Flow.

10.4 Bridging the Designer–Developer Vocabulary Gap

One of the less anticipated effects of EBD is the change it produces in cross-functional communication. Designers naturally think in journeys — the complete arc of what a user is trying to accomplish. Developers naturally think in components — the discrete units of rendering and behavior. These vocabularies are not wrong, but they are structurally misaligned, and the misalignment produces translation overhead in every

handoff. EBD provides a shared structural vocabulary: designers describe Experiences and Flows, developers implement Flows and Interactions, and both groups use the same terms to refer to the same architectural units. Design reviews become architecture reviews. The question shifts from “which screens does this feature touch?” to “which Flows does this change affect, and does the Experience need to know?”

10.5 Accessibility as a Structural Consequence

When Interactions are genuinely atomic — accepting typed input, rendering a bounded display state, emitting a well-defined event — they become naturally amenable to accessibility testing. Each Interaction has a clear input contract (what does the user provide?), a clear output contract (what event does it emit?), and a clear display state (what does the user perceive?). These contracts map directly to the concerns that accessibility standards address: labeling, focus management, keyboard operability, screen-reader announcements. Teams practicing EBD consistently report that accessibility compliance becomes easier not because they try harder, but because the structural unit they are testing is small enough to have an unambiguous accessibility contract. The improvement is architectural, not motivational.

10.6 The State Accumulation Pattern

Experiences that attempt to own the entire user session — rather than just the journey they represent — accumulate state that properly belongs elsewhere. An onboarding Experience begins tracking authentication tokens. A checkout Experience begins caching product catalog data. A settings Experience begins holding application-wide preferences. The result is an Experience that cannot be unmounted without losing unrelated application state, which defeats the composability promise of the tier. The correction is to distinguish journey state (which the Experience legitimately owns) from session state and application state (which belong in infrastructure layers outside the EBD hierarchy). An Experience should be mountable and unmountable without side effects beyond its own journey.

10.7 Configuration-Driven Composition as a White-Labeling Gateway

When Experience composition is driven by configuration rather than code, the distance between “different organizations see different flows” and “different organizations see a differently branded product” shrinks to a configuration delta. The same mechanism that selects which Flows appear in an Experience can select which theme, locale, and branding assets accompany them. Teams that adopt

configuration-driven composition for functional reasons — different user segments need different journeys — discover that they have inadvertently built the infrastructure for white-label deployment. The product becomes a composition engine; the configuration becomes the product definition. This is not a design goal of EBD per se, but it is a structural consequence that recurs frequently enough to warrant explicit acknowledgment.

10.8 The Refactoring Gradient

Teams rarely adopt EBD wholesale. More commonly, they begin by identifying one high-volatility journey in an existing codebase and restructuring it into the Experience-Flow-Interaction hierarchy while leaving the rest of the application untouched. This produces a hybrid codebase where some journeys are EBD-structured and others retain their original organization. The observation is that the hybrid state is stable and productive. The EBD-structured journeys become demonstrably easier to modify, and the contrast motivates incremental adoption. Attempting a full-codebase conversion in a single pass, by contrast, produces a long period of structural instability with no intermediate payoff. The refactoring gradient — one journey at a time, starting with the most volatile — consistently outperforms the big-bang approach.

11. Appendix: Case Study — Developer Onboarding Platform

This appendix presents a realistic application of EBD to a developer onboarding platform — a system that configures new developers' environments when they join an organization. The platform installs tools, provisions credentials, clones repositories, and establishes workspace structures. Different organizations require different onboarding sequences, and the platform must accommodate this variation without per-client code changes.

11.1 System Context

The platform serves multiple organizations. Each organization defines its onboarding requirements through a configuration package specifying which tools to install, which credentials to provision, which repositories to clone, and which workspace structures to create. Some organizations require all developers to complete every step; others make certain steps optional. Regulated industries may require additional compliance steps that do not appear in the standard sequence.

The interface must present the correct sequence of steps for each organization, collect the necessary input from the developer, execute the configuration, and report results — all without hardcoding any organization-specific logic into the interface layer.

11.2 Experience Decomposition

The top-level architectural unit is the **SetupExperience** — the complete onboarding journey from initial landing to a fully configured development environment. This Experience owns the journey lifecycle: it knows when onboarding begins, what the completion criteria are, and what happens when the journey is abandoned or resumed.

The SetupExperience composes four Flows, each owning a discrete goal:

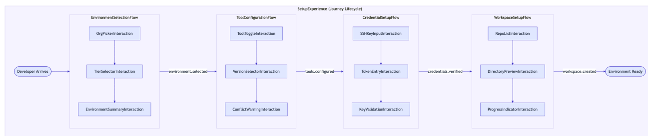
Flow	Goal	Completion Criteria
EnvironmentSelectionFlow	Identify the developer's target environment and organizational tier	Environment profile selected and confirmed
ToolConfigurationFlow	Select and configure the tools to be installed	Tool manifest finalized and validated
CredentialSetupFlow	Provision or collect authentication credentials (SSH keys, tokens, certificates)	All required credentials present and verified
WorkspaceSetupFlow	Clone repositories and establish the directory structure	All repositories cloned and workspace structure created

Each Flow is composed of Interactions:

Flow	Interactions
EnvironmentSelectionFlow	OrgPickerInteraction, TierSelectorInteraction, EnvironmentSummaryInteraction
ToolConfigurationFlow	ToolToggleInteraction, VersionSelectorInteraction, ConflictWarningInteraction
CredentialSetupFlow	SSHKeyInputInteraction, TokenEntryInteraction, KeyValidationInteraction
WorkspaceSetupFlow	RepoListInteraction, DirectoryPreviewInteraction, ProgressIndicatorInteraction

11.3 Core Journey

The following diagram illustrates the standard onboarding journey. The SetupExperience orchestrates the Flow sequence; each Flow manages its own Interactions internally.



diagram

11.4 Configuration-Driven Composition

The SetupExperience does not hardcode the Flow sequence. Instead, it reads the organization's configuration to determine which Flows to include, in what order, and with what parameters. A simplified configuration structure:

```yaml
onboarding:
  organization: "Acme Engineering"
  flows:
    - id: environment_selection
      required: true
      order: 1
    - id: tool_configuration
      required: true
      order: 2
    - id: credential_setup
      required: true
      order: 3
    - id: workspace_setup
      required: true
      order: 4
```

The SetupExperience reads this configuration at mount time, instantiates the corresponding Flows, and orchestrates their sequencing through the standard event-based protocol. It does not know what happens inside any Flow. It only knows that each Flow will eventually emit a completion event or an abandonment event, and it responds accordingly.

11.5 Adding a Compliance Flow

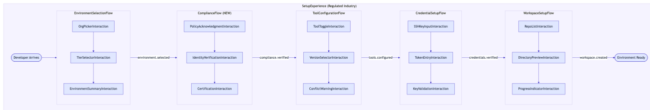
A regulated-industry client requires developers to acknowledge security policies, verify their identity through a corporate identity provider, and complete a data-handling certification before any tools are installed. In a screen-organized codebase, this would require modifying navigation logic, inserting conditional steps into existing wizards, and threading new state through the entire onboarding sequence.

In the EBD-structured system, the change is:

1. **Create a new Flow**: `ComplianceFlow`, composed of `PolicyAcknowledgmentInteraction`, `IdentityVerificationInteraction`, and `CertificationInteraction`.
2. **Update the configuration** for regulated-industry organizations:

```yaml
onboarding:
  organization: "Regulated Corp"
  flows:
    - id: environment_selection
      required: true
      order: 1
    - id: compliance
      required: true
      order: 2
    - id: tool_configuration
      required: true
      order: 3
    - id: credential_setup
      required: true
      order: 4
    - id: workspace_setup
      required: true
      order: 5
```

No existing Flow is modified. The EnvironmentSelectionFlow, ToolConfigurationFlow, CredentialSetupFlow, and WorkspaceSetupFlow are identical to their non-regulated counterparts. The SetupExperience renders the ComplianceFlow between environment selection and tool configuration because the configuration says to — not because any code was changed.



diagram

11.6 Structural Observations

This case study illustrates several properties of EBD in practice:

- **Additive change model**: New organizational requirements produce new Flows and configuration entries, not modifications to existing Flows. The blast radius of a new requirement is bounded by design.
- **Flow independence**: Each Flow is testable in isolation. The ToolConfigurationFlow can be mounted outside the SetupExperience, provided with mock input, and validated against its completion contract without any other Flow present.
- **Interaction reuse**: The ProgressIndicatorInteraction appears in the WorkspaceSetupFlow but could appear in any Flow that needs to report long-running progress. Its contract — accept a progress payload, render a visual state, emit a cancellation event if requested — is independent of the Flow that hosts it.
- **Configuration as product definition**: The difference between two organizations' onboarding experiences is fully expressed in their configuration packages. The codebase is shared infrastructure. The configuration is the product.

- **Compliance as composition**: Regulatory requirements, which in many systems trigger invasive cross-cutting modifications, become compositional in EBD. A compliance step is a Flow. Including it is a configuration decision. Excluding it is a configuration decision. The code does not know whether it is operating in a regulated context or not — only the configuration does.

✦ ✦ ✦

12. Conclusion

The structural failures most visible in interface codebases — navigation logic in leaf components, flow state in global stores, cascading changes from localized decisions — share a common origin. The codebase is organized around what the system presents rather than what the user intends to accomplish. When the organizing unit and the unit of human purpose are misaligned, every structural decision is made in the wrong frame of reference.

Experience-Based Decomposition closes that gap. By establishing experiences, flows, and interactions as distinct structural roles with clear responsibilities and explicit communication rules, it makes user intent legible in the code itself. The developer reading the codebase can trace a user journey from its initiation to

its conclusion without reconstructing it from navigation conditionals distributed across dozens of components.

In practice, interfaces organized this way do not eliminate change. Configurations change. Organizational requirements change. Products evolve. What changes is the experience of making those changes: a new field type affects one interaction component. A new conditional flow affects one experience and the configuration schema. A new audience segment produces a different experience composition without touching existing code.

When the communication model is event-based and the composition model is configuration-driven, something further becomes true: the interface layer itself becomes a deployment artifact, not just a code artifact. Experiences can be composed, ordered, and conditioned through configuration managed outside the release cycle. The distance between an organizational decision and its expression in the user's experience shrinks to the propagation delay of a configuration update.

That is the intended destination — not merely an interface that is easier to change, but one where the relevant changes no longer require code.

✦ ✦ ✦

Appendix A: Glossary

Environmental Boundary (Experience) — The Experience tier is the exclusive communicator with the backend API. Environmental volatility — API contract changes, backend migrations, authentication changes — is absorbed at this boundary so that Flow and Interaction logic remains unaffected.

Communication Rules — The directional constraints governing how EBD tiers interact: Experiences invoke Flows, Flows invoke Interactions, and no component may call a peer at its own tier. These rules preserve encapsulation and make change impact predictable.

Configuration-Driven Composition — The practice of defining which flows an experience contains, and in what order, through external configuration rather than hardcoded structure. Enables organizational decisions to reshape user journeys without code changes.

Core User Journey — A high-level human purpose defining the primary value of the interface. Used as an architectural validation mechanism to ensure decomposition boundaries align with real user intent.

Cross-Cutting Volatility — Change that affects capabilities spanning all tiers, such as logging, authentication, theming, or analytics. Isolated into

Utilities so that a single cross-cutting change does not require modifications across Experiences, Flows, and Interactions.

Environmental Volatility — Change originating outside the interface layer, including API contract revisions, backend migrations, and deployment-target differences. Contained by the Experience tier, which is the exclusive communicator with the backend, so that Flow and Interaction logic remains unaffected.

Experience — A complete, composable user journey bounded by a human intent. The most stable structural tier in EBD. An Experience assembles Flows through configuration and manages journey state across them.

Experience-Based Decomposition — A volatility-based decomposition approach for user interfaces that organizes code into three structural tiers — Experience, Flow, and Interaction — aligned with the axes along which interface change actually occurs. Sibling framework to Volatility-Based Decomposition (VBD).

Flow — A goal-directed sequence of interactions accomplishing one discrete outcome within an experience. The middle tier. Owns accumulated state and skip conditions; does not make API calls (that responsibility belongs to the Experience).

Flow Composition — The mechanism by which an Experience assembles its constituent Flows, typically through external configuration that specifies flow identity, ordering, and skip conditions. Decouples journey structure from release cycles.

Functional Volatility — Change in business logic within Flows, such as new validation rules, altered calculation steps, or revised decision criteria. Contained at the Flow tier so that structural and environmental boundaries remain unaffected.

Interaction — An atomic, observable user act. The most volatile tier. Owns only its display state and the event it emits upward to its parent Flow.

Interaction Event — The atomic event an Interaction emits upward to its parent Flow upon completion of a user act. Interaction Events are the sole communication mechanism from the Interaction tier to the Flow tier, preserving directional control.

Interface Volatility Axis — A dimension along which interface change is expected: functional, non-functional, cross-cutting, or environmental. EBD aligns architectural boundaries with these axes so that a change along one axis does not ripple across others.

Journey State — The accumulated data that an Experience maintains as the user progresses through its Flows. Each Flow contributes to journey state upon completion; no Flow reads or mutates another Flow's internal state directly.

Peer Prohibition — The rule that components at the same structural tier must not invoke or directly communicate with one another. Flows do not call Flows; Interactions do not call Interactions. This constraint ensures that each component can be added, removed, or reordered without side effects on its siblings.

Skip Condition — A configuration-driven rule that determines whether a particular Flow is included or bypassed within an Experience. Skip conditions allow the same Experience definition to produce different journey shapes based on organizational context, locale, or feature flags.

Non-Functional Volatility — Change driven by the reorganization of user journeys, such as reordering steps, adding or removing Flows, or splitting an Experience into variants. Contained at the Experience tier through configuration-driven composition.

Tier — One of the three structural levels in EBD — Experience, Flow, or Interaction — each corresponding to a distinct grain of user intent and a distinct volatility profile. The tier hierarchy defines both the communication direction and the scope of permissible change.

Utility — A cross-cutting interface capability consumed across all tiers without domain-specific knowledge. Utilities encapsulate concerns like theming, logging, and authentication that would otherwise duplicate across the tier hierarchy.

✦ ✦ ✦

Appendix B: Applicability Checklist

Experience-Based Decomposition is well-suited for interfaces that:

- Serve more than one audience type with distinct mental models or organizational contexts
- Contain multi-step workflows with conditional paths or configurable step presence
- Must support multiple locales, themes, or organizational branding configurations
- Are driven by external configuration that determines which features or flows appear
- Will be maintained across multiple years and multiple product teams
- Depend on backend systems that may route differently based on the same configuration the interface uses

It may be disproportionate for:

- Single-purpose utility screens with no conditional logic
- Short-lived interfaces with a bounded, stable feature set
- Systems where the expected rate of structural change does not justify the decomposition overhead

✦ ✦ ✦

References and Influences

William Christopher Anderson Anderson, William Christopher. *Volatility-Based Decomposition in Software Architecture.* February 2026.

Experience-Based Decomposition is a direct sibling framework to VBD. VBD established that architectural boundaries aligned with volatility axes produce systems resilient to change. EBD applies the same organizing principle to the interface layer, extending the Manager-Engine-Accessor-Utility taxonomy into Experience-Flow-Interaction-Utility, and establishing the same communication discipline — directional, role-constrained, event-based — at the UI level. The configuration-driven composition model described in Section 6 is explicitly designed to compose with VBD's manager-level routing: the same

organizational configuration that determines which flows appear in the interface determines which backend paths execute in response to those flows completing.

Alan Cooper, Robert Reimann, David Cronin, Christopher Noessel Cooper, Alan et al. *About Face: The Essentials of Interaction Design.* Fourth Edition. Wiley, 2014.

Cooper's goal-directed design methodology established the foundational claim that interfaces should serve user goals, not reflect system capabilities. His use of personas and scenarios to drive interaction decisions is a direct precursor to the core user journey validation mechanism in EBD. Cooper also articulated the distinction between implementation models (how systems work) and mental models (how users think systems work), which maps directly to the argument in Section 1 that organizing code around screens misaligns it with the user's cognitive frame. EBD can be understood as the architectural expression of Cooper's design-level insight.

Brad Frost Frost, Brad. *Atomic Design.* Brad Frost, 2016.

Atomic Design established a hierarchical vocabulary for component composition — atoms, molecules, organisms, templates, pages — and demonstrated that interface structure benefits from explicit layering. EBD inherits this hierarchical instinct

while operating at a different granularity. Where Atomic Design hierarchizes visual structure, EBD hierarchizes behavioral intent. The two frameworks are complementary: Atomic Design governs the construction of interaction components; EBD governs how those components are assembled into purposive sequences. A ProductCard (interaction, in EBD terms) might itself be composed of atoms and molecules (in Atomic Design terms). Neither taxonomy replaces the other.

Dan Saffer Saffer, Dan. *Microinteractions: Designing with Details.* O'Reilly Media, 2013.

Saffer's analysis of microinteractions — the small, contained moments that define product character — provides the conceptual foundation for the Interaction tier in EBD. His four-part structure (trigger, rules, feedback, loops and modes) maps closely to EBD's definition of an interaction as an atomic act with its own display state and event output. Saffer's central argument, that interactions deserve dedicated design attention rather than being treated as implementation details, reinforces the structural case for isolating them in their own tier rather than embedding them in flow or experience logic.

Jesse James Garrett Garrett, Jesse James. *The Elements of User Experience.* Second Edition. New Riders, 2010.

Garrett's five-plane model — strategy, scope, structure, skeleton, surface — introduced the idea that user experience design has distinct layers of abstraction, each with its own concerns and each dependent on decisions made in the layers below. EBD is an architectural expression of the same layering intuition, applied specifically to the structure and skeleton planes. Garrett's framework is primarily a design process model; EBD converts the same insight into an implementation model, giving the structural layer a home in the codebase rather than just in the design process.

Donald A. Norman Norman, Donald A. *The Design of Everyday Things.* Revised and Expanded Edition. Basic Books, 2013.

Norman's articulation of affordances, feedback, and the gulf of evaluation provided the cognitive psychology grounding for interaction design as a discipline. His concept of the gulf between the system image and the user's mental model is directly relevant to the argument in Section 2 that screen-organized code produces a codebase whose structure is legible to developers but not to the human intent it serves. EBD attempts to close the same gulf at the architectural level — making the code's structure reflect the user's purpose rather than the system's structure.

Kim Goodwin Goodwin, Kim. *Designing for the Digital Age: How to Create Human-Centered Products and Services*. Wiley, 2009.

Goodwin's comprehensive treatment of goal-directed design in practice — including scenario development, flow mapping, and design framework construction — provides the practitioner vocabulary that EBD formalizes architecturally. Her emphasis on designing for people's goals across time (not just in a single interaction) reinforces the experience tier's role as the durable container of user intent. Her scenario-based validation approach is the design-side analog to EBD's core user journey validation mechanism.

Roy Thomas Fielding Fielding, Roy Thomas. *Architectural Styles and the Design of Network-Based Software Architectures*. Doctoral dissertation, UC Irvine, 2000.

Fielding's REST architectural style demonstrated that well-defined constraints on communication between components — stateless requests, uniform interface, layered system — produce distributed systems that evolve gracefully. The communication rules in EBD apply the same discipline to the interface layer: constrained, directional, role-specific communication prevents the coupling that undermines architectural integrity. The event-based communication model in Section 6 draws on the same

insight — decoupling the interface from the backend through a uniform event interface rather than tight procedural coupling.

Gregor Hohpe and Bobby Woolf Hohpe, Gregor; Woolf, Bobby. *Enterprise Integration Patterns.* Addison-Wesley, 2003.

Hohpe and Woolf's catalog of messaging patterns provides the foundational vocabulary for the event-based communication model described in Section 6. Their treatment of message channels, event-driven routing, and the decoupling of producers from consumers is directly applicable to the mechanism by which flow completion events propagate to both the experience tier above and the backend manager below. The configuration-driven routing described in that section is an application of their correlation and content-based routing patterns to the interface-backend boundary.

Juval Löwy Löwy, Juval. *Righting Software.* Addison-Wesley, 2019.

Löwy's IDesign methodology, the source of Volatility-Based Decomposition, provides the structural vocabulary from which EBD draws its parallel taxonomy. The Manager-Engine-Accessor-Utility role taxonomy and the associated communication rules are adapted here into Experience-Flow-Interaction-Utility. Löwy's emphasis on separating orchestration from execution — and his

warning that interwoven orchestration and execution logic creates change coupling — translates directly to the EBD rule that experience logic (orchestration) must be structurally separated from flow logic (execution of a specific goal).

✦ ✦ ✦

Author's Note

Experience-Based Decomposition builds on established practices in interaction design, frontend engineering, and software architecture — in particular Alan Cooper's goal-directed design, Brad Frost's Atomic Design, and the volatility-first thinking that underlies Volatility-Based Decomposition. Whether the specific synthesis presented here is genuinely novel, the author cannot say with certainty. What can be said is that the author has not encountered another framework that approaches interface decomposition this way, nor read work that expresses this structural model — the explicit tier hierarchy, the communication rules, the configuration-driven composition model, and the deliberate isomorphism with the backend component taxonomy. If prior art exists, the author would welcome the reference.

The intent is a reference suitable for design reviews, engineering onboarding, and architectural decision-making in organizations building products intended to serve users well over years rather than quarters.

✦ ✦ ✦

Distribution Note

This document is provided for informational and educational purposes. It may be shared internally within organizations, used as a reference in product design and engineering discussions, or adapted for non-commercial educational use with appropriate attribution.

APPENDIX D

Boundary-Driven Testing

Boundary-Driven Testing (BDT)

Deriving Test Strategy from Architectural Boundaries: A Practitioner-Oriented Articulation

Author: William Christopher Anderson
Date: April 2026
Version: 1.0

✦ ✦ ✦

Executive Summary

Testing difficulty is architectural evidence. When a component cannot be exercised in isolation, when a unit test requires a running database, when a change in one module breaks tests across a dozen others —

the problem is not the tests. It is the structure that the tests are attempting to exercise. The component has no coherent boundary. Its responsibilities are distributed incorrectly. Its dependencies are hidden. Tests fail not because the code is wrong, but because the code was not organized to be observable or controllable in the first place.

Boundary-Driven Testing is the observation that the test spiral — unit, integration, end-to-end, system, user acceptance — is not a testing methodology. It is an architectural map. Each ring of the spiral corresponds to a level of architectural scope, and that scope is determined entirely by where boundaries have been placed. Get the boundaries right and the spiral populates itself: each tier has clear targets, predictable scope, and low maintenance overhead. Get them wrong and the spiral collapses — unit tests become integration tests in disguise, E2E tests become the only reliable safety net, and the entire suite grows expensive while providing diminishing confidence.

The structural models defined in *Volatility-Based Decomposition* and *Experience-Based Decomposition* produce boundaries that localize both change and test scope simultaneously. The same line that prevents coupling prevents test contamination. The same role taxonomy that makes components replaceable makes them mockable. The same core use cases that validate

structural boundaries generate test scenarios across the full spiral. This is not a coincidence. It is the intended consequence of decomposing correctly.

✦ ✦ ✦

Abstract

The relationship between architectural structure and testability is direct and bidirectional: clear boundaries produce testable components, and testing difficulty reveals boundary problems. Boundary-Driven Testing articulates this relationship by mapping the test spiral onto the component role taxonomy defined in Volatility-Based Decomposition (VBD) and Experience-Based Decomposition (EBD). Each role — Manager, Engine, Resource Accessor, Utility at the system level; Experience, Flow, Interaction, Utility at the UX level — has a natural test profile determined by its responsibilities, permitted dependencies, and communication rules. Mock placement, test scope, and assertion strategy follow from structural position. The spiral is a structural mirror. Difficulty at any level of the spiral points to a specific class of boundary problem — and to the structural fix.

✦ ✦ ✦

1. Introduction

The conventional framing of testing as a discipline separate from design produces a particular kind of pain. Teams adopt frameworks, mandate coverage minimums, and write guidelines about what to test. The tests improve. The pain persists. A change to a business rule breaks seventeen tests, most of which are not about business rules. An integration test requires spinning up four services to assert one value. An end-to-end test passes in isolation and fails in CI for reasons nobody can reproduce. More coverage, more pain.

The reframe is simple but consequential: **testing is not separate from design.** The structure of the system determines what can be tested and how. A component designed around a coherent responsibility, with explicit inputs, explicit outputs, and dependencies passed rather than acquired, is inherently testable. No additional effort is required to make it so. The same structural choices that allow the component to change without cascading effects allow it to be tested without elaborate setup.

The inverse is equally true. A component that cannot be unit-tested without mocking half the system is not badly tested — it is badly structured. The test difficulty is diagnostic. It reveals that the component has absorbed responsibilities it should not have, or that its dependencies are implicit rather than declared,

or that the boundary between it and its collaborators has been drawn in the wrong place. Fix the structure; the tests follow. Paper over the structure with more elaborate test scaffolding; the problem remains and compounds.

Boundary-Driven Testing takes the diagnostic function of tests seriously. It maps the test spiral to the structural models defined in VBD and EBD, making explicit which components belong at which level of the spiral and why. It treats mock placement as architectural evidence — you mock at boundaries, and if you are mocking everywhere, you have too many boundaries or they are in the wrong places. And it articulates the consequence of correct decomposition: a test suite that is fast at the base, targeted in the middle, and confident at the top, maintained by the natural structure of the system rather than by constant manual curation.

✦ ✦ ✦

2. The Spiral Is a Structural Map

The test spiral describes a progression from the narrowest to the broadest scope:

- **Unit** — one component, all dependencies replaced. Fast, numerous, fine-grained.

- **Integration** — component collaboration across one seam, with dependencies mocked at the outer boundary. Verifies that orchestration logic and contracts are wired correctly.
- **End-to-End** — a complete user flow, full stack and browser. Verifies that the system behaves correctly from the outside.
- **System** — the integrated system under realistic conditions: load, failure injection, configuration variation. Verifies non-functional qualities.
- **User Acceptance** — real users or proxies confirm that what was built matches what was intended.

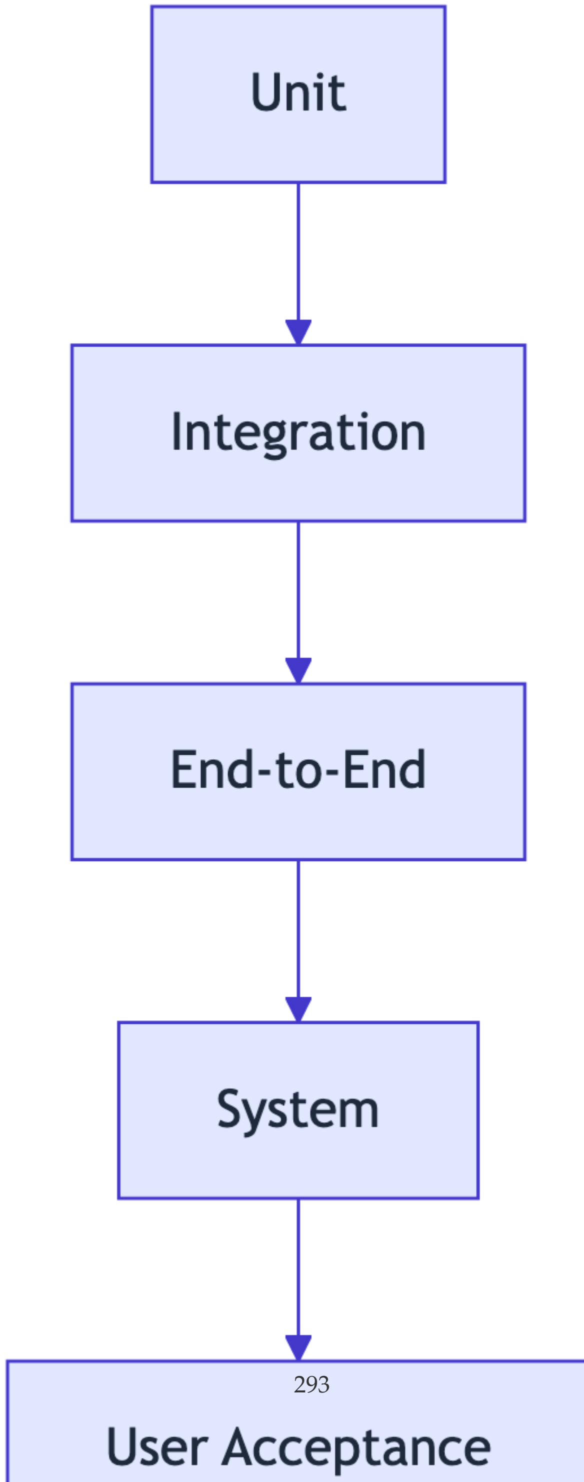
The common reading of the spiral is proportional: many unit tests, fewer integration tests, fewer still E2E, and so on. This is a useful heuristic, but it obscures the more important principle. The spiral is not primarily about proportion — it is about scope. Unit scope is a single component. Integration scope is a collaboration across one boundary. E2E scope is a complete journey. System scope is the whole.

Scope is determined by architecture. Where boundaries are placed determines what constitutes a "unit," what constitutes an "integration," and what constitutes a "journey." In a system with no meaningful component boundaries, unit scope and system scope are the same thing — there is nothing

below the full system that can be isolated. The spiral collapses into E2E by default, because E2E is the only level at which you can exercise anything coherent.

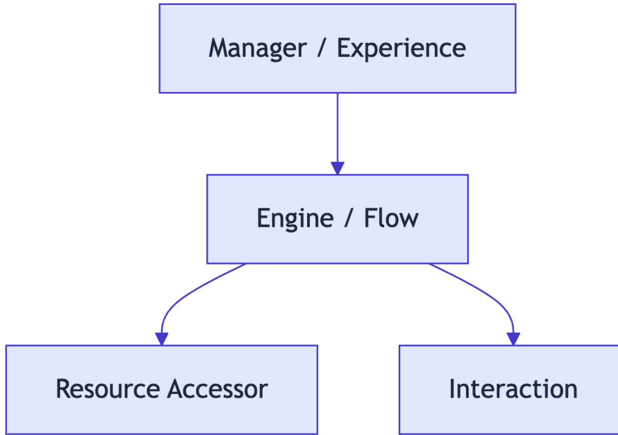
Figure 1a shows the five levels of the spiral from narrowest to broadest scope. Figure 1b shows how each architectural tier attracts tests at a specific level.

Figure 1a — The Test Spiral


Unit
Integration
End-to-End
System
User Acceptance

diagram

Figure 1b — Architectural Tier → Test Level



diagram

✦ ✦ ✦

3. Boundaries Determine Test Profiles

Each component role in VBD and EBD has a characteristic test profile — not assigned arbitrarily, but derived from the role's structural position, responsibilities, and communication rules.

3.1 Engines — The Unit Test Core

Engines are the most logic-dense tier and the natural home of the unit test suite. An Engine encapsulates business rules: given inputs, apply policy, produce a result. It has no workflow awareness, no sibling dependencies, and no reason to reach outward unless it needs data from a Resource Accessor — which it receives through an explicit, mockable interface.

This is what makes Engines straightforwardly testable. Mock the Accessor, supply controlled inputs, assert on the output. The Engine's communication constraints — no peer Engine calls, no direct infrastructure access — ensure there is nothing else to mock. The test scope is exactly the Engine and nothing more.

For Flows in EBD the same principle applies. A Flow receives shared state from the Experience, steps through Interactions, accumulates state, and emits a completion event upward. Flows do not make backend calls — that is the Experience's exclusive responsibility. Simulate Interaction events through a test harness and assert on what the Flow emits as accumulated state. No API mocking is needed; the Flow has no external dependencies. The Flow's rule against calling sibling Flows means the unit scope stays tight.

3.2 Resource Accessors — Thin Boundary, Minimal Unit Surface

Accessors sit at the system's external boundary, and that position defines what they are responsible for testing — which is less than it might appear. An Accessor's job is translation: convert a domain request into an external call, convert the response back. Whether the external system is reachable, whether it is correctly provisioned, whether it performs within acceptable bounds — none of these are Accessor concerns. They are infrastructure concerns, and they belong to system testing and deployment verification.

If an Accessor contains meaningful translation or mapping logic, that logic can be unit tested by controlling the inputs and outputs through the Accessor's own interface. But the Accessor has no business connecting to a real database in a unit or integration test. It either connects or it doesn't — and that is a system-level fact, not a test target. The Accessor's correctness is about the translation. The infrastructure's correctness is about the infrastructure.

3.3 Integration Tests — The Three Seams That Matter

Integration tests in a VBD system are not about any single component in isolation. They are about the *seams* between roles — verifying that the contracts components depend on are honored, and that

orchestration logic matches the design. Everything is still mocked at the external boundary. Real external systems do not enter the picture until E2E.

The distinction from unit tests is scope, not realism. A unit test exercises one component against mocked dependencies. An integration test exercises the *collaboration* between two components against mocked dependencies at the outer edge. You care whether the wiring is correct — not whether the database is running.

There are three seams worth testing at this level:

Manager → Engine. Does the Manager invoke the Engine with the correct inputs? Does it handle every state the Engine's contract can emit — success, domain failure, unexpected error — and route accordingly? The Engine is mocked. Feed it controlled responses representing each state it might return. Verify that the Manager's orchestration logic handles all of them correctly. A mock Engine returning a validation failure is just as useful as a real one — what you are testing is the Manager's response, not the Engine's behavior.

Engine → Resource Accessor. Does the Engine correctly use the Accessor's contract? Does it correctly interpret the states the Accessor can return? The Accessor is mocked. No database is involved. You are testing whether the Engine handles the Accessor's interface correctly — not whether the Accessor connects to anything.

Manager → Resource Accessor. Managers sometimes interact with Accessors directly — for reads that inform orchestration decisions, or for state persistence the Manager owns. Test these paths the same way: mock the Accessor, exercise the Manager's handling of every response state the Accessor's contract defines.

In EBD, the equivalent seam is Experience → Flow: does the Experience pass correct shared state, handle Flow completion and skip signals, and advance the journey correctly? The backend is mocked. You are verifying journey composition logic — not backend behavior.

3.4 Interactions and Utilities — Narrow and Fast

Interactions are atomic. They render, receive user input, and emit events. They carry no flow logic, make no API calls, and have no awareness of adjacent components. Component tests — render in a harness, simulate the input event, assert what was emitted — cover them completely and quickly. No mocks are typically needed; props and callbacks are the entire interface.

Utilities are simpler still: inputs in, outputs out, no side effects. Given input X, assert output Y. The only exception is a Utility wrapping an external sink (a log transport, a telemetry exporter), where the sink gets mocked. Everything else is pure function territory.

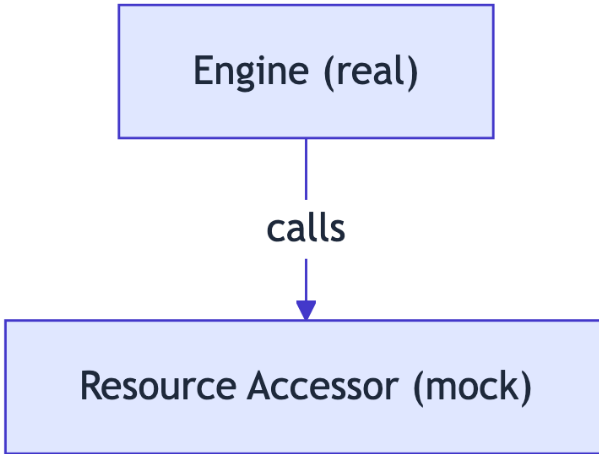
4. Mock Placement Is Architectural Evidence

Where you place mocks tells you where your boundaries are. Where you are forced to place mocks tells you where your boundaries should be.

The rule is simple: mock at the role boundary, not inside the role. Each component role has one natural mock point — the interface at which it hands off to the next tier. Mock that interface and nothing else.

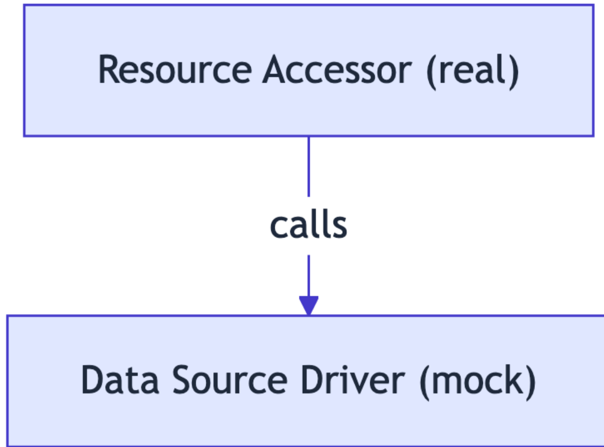
Figure 2 shows where mocks belong at each test level. Each diagram is independent — together they cover the full VBD test surface.

Figure 2a — Engine unit test: mock the Resource Accessor, keep the Engine real.



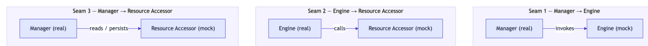
diagram

Figure 2b — Accessor unit test: mock the data source, keep the Accessor real.



diagram

Figure 2c — Integration tests: the three seams. Each seam tests one collaboration. Dependencies at the outer edge are mocked — no real external systems.



diagram

When a unit test requires mocking more than the single boundary below the component under test, something is wrong. Either the component has absorbed responsibilities that belong at a different tier, or its dependencies are implicit rather than injected, or

an Accessor is missing and the component is reaching directly into infrastructure it should not see. Mock proliferation is always a structural signal — not a testing problem, and not a problem that better mocking frameworks solve.

The inverse is equally worth examining. An Engine or Flow unit test that requires no mocks is either genuinely pure-computation (rare and fine) or is only exercising the easy path through logic that silently delegates to collaborators the test never reaches. Coverage numbers tell you how many lines ran. They do not tell you whether the logic that matters was actually exercised.

✦ ✦ ✦

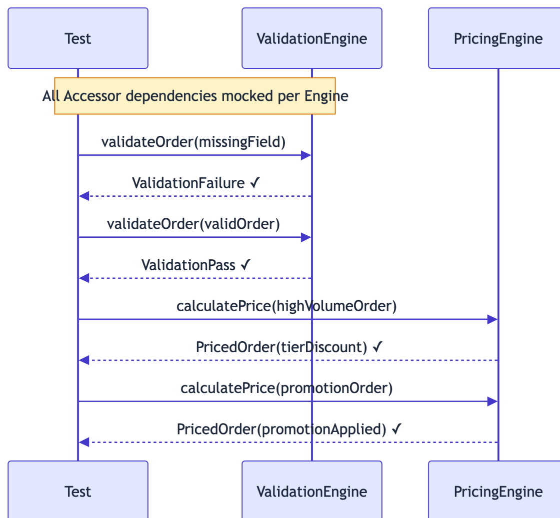
5. The Same Scenarios Validate Architecture and Tests

VBD and EBD both use core scenarios as architectural validation mechanisms. A core use case in VBD — Process an order, Evaluate eligibility, Onboard a new customer — should be traceable through the component hierarchy without bypassing communication rules. A core user journey in EBD — Complete developer onboarding, Publish a prism, Discover a workspace — should trace through Experience → Flow → Interaction without boundary leakage.

These scenarios are also the test scenarios that matter most. Not because coverage demands it, but because scenarios that validate structural boundaries naturally exercise the most load-bearing code paths, the most significant collaborations, and the most complete representations of what the system is actually for.

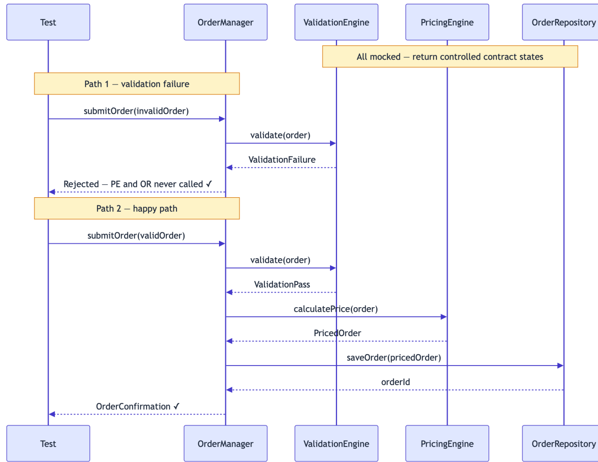
Figures 3a, 3b, and 3c show the same order-processing scenario at three levels of the spiral. Each level asks a different question. Each has a different scope.

Figure 3a — Unit: each Engine in isolation



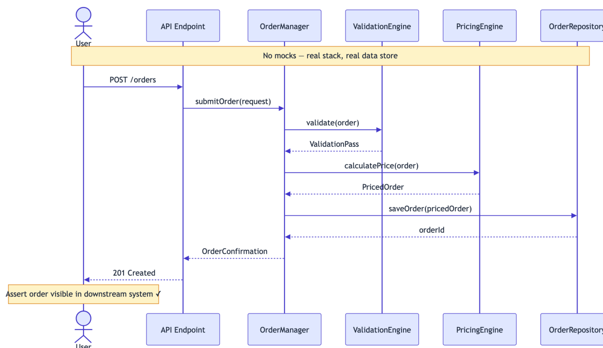
diagram

Figure 3b — Integration: Manager orchestration with mocked dependencies



diagram

Figure 3c — E2E: full stack, real systems



diagram

The same scenario, three questions:

- **Unit** — Does ValidationEngine correctly reject a missing field? Does PricingEngine apply the right discount? One Engine, all its rule branches, mocks for anything it depends on.
- **Integration** — Does OrderManager route correctly when validation fails? Does it call the right dependencies in the right order when it succeeds? Engines and Accessor are mocked — you are testing the Manager's orchestration logic against every contract state its dependencies can emit.
- **End-to-End** — Does an order submitted through the real API surface in the system correctly? No mocks. Real behavior, real infrastructure, real assertion.
- **User Acceptance** — Does a stakeholder placing an order through the application experience the outcome they expected?

Each level asks a different question about the same scenario. Each question corresponds to a structural scope. The test suite is not organized around coverage targets — it is organized around the architecture.

When a scenario cannot be cleanly decomposed this way — when the unit tests would require mocking the Manager, or the integration tests have no obvious boundary to stop at — the scenario is

exposing an architectural gap. The fix is structural. Tests that are uncomfortable to write at a given level signal that the corresponding structural tier is missing or muddled.

✦ ✦ ✦

6. UAT Validates What Architecture Validated First

User acceptance testing is often treated as a separate world from the structural concerns of the preceding spiral levels. Stakeholders exercise the system. They are not concerned with Managers, Engines, or Flows. They care whether the product works as intended.

But the core user journeys that structure EBD — and the core use cases that structure VBD — are precisely the scenarios that UAT exercises. The developer onboarding journey that validates EBD structural boundaries is the same journey that a UAT participant walks through. The order processing scenario that exercises VBD communication rules is the same scenario a business stakeholder confirms in acceptance.

This alignment is not accidental. Both architectural validation and UAT begin from the same question: does the system fulfill its core purpose correctly? Architectural validation asks it structurally — can this

scenario be traced without boundary violations? UAT asks it experientially — does this scenario produce the correct outcome for a real user?

When the architecture is sound, the answers converge. Structural boundaries support real journeys without friction. UAT scenarios map cleanly onto the E2E scenarios that confirm those journeys in the automated suite. The test spiral closes: the same scenarios that entered at the unit level — as isolated assertions on Engine logic — emerge at the UAT level as confirmed product behavior.

When they diverge — when UAT surfaces scenarios that have no corresponding structural representation, or when E2E tests cover journeys that no stakeholder actually cares about — both the tests and the architecture need reexamination.

✦ ✦ ✦

7. Diagnostic Signals

Testing difficulty is a signal. The nature of the difficulty points to the specific structural problem.

Mock proliferation — a unit test mocking more than one or two dependencies is usually exercising something that spans too many concerns. The component should be decomposed, or its dependencies should be consolidated behind a single interface.

Slow unit tests — unit tests that require real I/O (network, filesystem, database) are not unit tests. Something that should be an Accessor is embedded in an Engine or Flow. Extract it.

Brittle E2E tests — E2E tests that break for reasons unrelated to user-visible behavior are coupled to implementation detail. Either the test is asserting on internal component state (stop), or the flow being tested has no stable boundary (fix the architecture).

Inverted pyramid — when the E2E suite is larger than the unit suite because unit tests cannot cover meaningful scenarios, the unit-testable tiers (Engines, Flows) contain less logic than they should. Business logic has migrated into Managers or Accessors.

UAT surprises — when UAT surfaces behaviors that no automated test predicted, either the core scenario set is incomplete or the structural model does not reflect how the product is actually used. Both are architectural discoveries, not testing failures.

✦ ✦ ✦

7A. Practitioner Observations

The following observations emerge from applying Boundary-Driven Testing across multiple systems of varying scale and domain. They are not prescriptive rules but recurring patterns — structural phenomena

that practitioners encounter once the spiral is treated as an architectural map rather than a coverage checklist.

The Test Migration Pattern

When architecture improves — when an Engine is extracted from a Manager, or an Accessor is separated from inline infrastructure calls — tests naturally migrate from integration scope to unit scope. Logic that previously could only be exercised through the Manager's orchestration can now be tested directly against the extracted Engine. The test count at each spiral level is therefore a leading indicator of structural health. A system with a growing unit test count and a shrinking integration test count is decomposing correctly: logic is moving into testable tiers. A system where unit tests plateau while integration tests multiply is accumulating orchestration logic in the wrong places. Tracking this ratio over time reveals architectural trajectory more reliably than any static metric.

The Mock Boundary Audit

Periodically reviewing where mocks are placed across the test suite reveals architectural drift before it becomes visible in production behavior. When a unit test that once required a single mock — the Accessor interface below the Engine — now requires three

mocks, a boundary has leaked. The Engine has acquired a dependency it should not have, or a new collaborator has been introduced without going through the established interface. The audit is mechanical: list every mock in every unit test, group by component under test, and compare to the expected mock count for that component's role. Engines should mock Accessors. Managers should mock Engines and Accessors. Utilities should mock nothing or at most one external sink. Deviations from this pattern are not judgment calls — they are structural findings that point to specific refactoring targets.

The E2E Stability Correlation

E2E test stability correlates directly with Experience and Manager stability. When E2E tests become flaky — passing on one run, failing on the next, sensitive to timing or environment — the instability almost always traces to the orchestration tier rather than to infrastructure. The Manager or Experience is making decisions that depend on transient state, or it is sequencing operations in a way that is sensitive to timing that unit and integration tests never exercise. Stable Managers produce stable E2E tests. When E2E flakiness spikes, the first diagnostic step is not to add retries or increase timeouts — it is to examine what changed in the orchestration layer. The E2E suite is a Manager health monitor.

The Coverage Paradox

Teams with high line coverage but poor boundary coverage consistently have worse defect rates than teams with moderate line coverage and strong boundary coverage. The explanation is structural: line coverage rewards exercising code paths, but many code paths are internal to a component and exercise only the easy branches. Boundary coverage — ensuring that every contract state a dependency can emit is handled by the caller — exercises the load-bearing logic: error handling, fallback paths, state transitions that only occur when a collaborator returns something unexpected. A team at 90% line coverage that never tests what happens when the Accessor returns a timeout has covered the lines but missed the boundary. A team at 65% line coverage that tests every Accessor contract state — success, not-found, timeout, malformed response — has covered less code but exercised far more of what matters. What matters is exercising contract states, not line counts.

Accelerated Test Review

BDT makes test review faster and more consistent because reviewers have a structural question to ask rather than a subjective one. Instead of evaluating whether "enough" is tested or whether the test "looks right," a reviewer checks that the test targets the correct spiral level for the component's role. Is this an

Engine? Then the test should be a unit test with mocked Accessors. Is this a Manager? Then the test should be an integration test exercising orchestration against mocked Engines. Is this a new E2E scenario? Then it should trace a complete user journey through the real stack. The review question shifts from “is this a good test?” to “is this test at the right level?” — a question with a definitive answer derived from the component’s structural position. Review time drops because the evaluation criteria are objective.

Natural CI Pipeline Mapping

BDT interacts with CI pipelines by providing a principled mapping between spiral levels and pipeline stages. Unit tests run on every commit — they are fast, numerous, and catch logic regressions immediately. Integration tests run on pull request — they verify that the collaboration contracts between components are intact before code enters the shared branch. E2E tests run on merge to main — they confirm that the full system behaves correctly before deployment. System tests run in staging — they exercise non-functional qualities under realistic conditions. The spiral maps to the pipeline stages naturally because each stage has a different latency tolerance and a different confidence target. Teams that adopt BDT often find that their pipeline stage definitions, which previously felt

arbitrary, now have a structural justification: each stage corresponds to a scope, and each scope corresponds to a tier.

The Diagnostic Cascade

A failing E2E test should be reproducible as a failing integration test at one specific seam. If it is, the defect is localized: the collaboration between two components at that seam is broken, and the integration test pinpoints which contract state is mishandled. If it is not — if the E2E test fails but all integration tests pass — then one of two things is true. Either the E2E test is coupled to implementation detail that no integration test targets (the test is wrong), or the integration test suite is missing a contract state that the real system exercises (the suite is incomplete). The diagnostic cascade — E2E failure, then integration reproduction, then unit isolation — is how BDT converts a symptom into a structural finding. Each level of the cascade narrows the scope. A defect that survives to E2E without appearing at integration is always an architectural discovery about a missing seam or an untested contract state.

The Refactor Safety Observation

When a system is structured according to VBD and tested according to BDT, internal refactoring within a component — changing how an Engine computes a

result, optimizing an Accessor's query strategy, restructuring a Manager's orchestration sequence — is protected by the correct test level automatically. Refactoring an Engine's internals is covered by its unit tests. Changing an Accessor's implementation is covered by its unit tests against the mocked data source. Altering a Manager's orchestration sequence is covered by its integration tests against mocked Engines. The key observation is that tests at the boundary do not care about internal restructuring — they care about the contract. This means that well-placed boundary tests provide refactoring confidence without requiring test updates for internal changes. When a refactor requires updating tests at multiple spiral levels, it is not a refactor — it is a contract change, and the test updates are the correct response.

✦ ✦ ✦

8. Conclusion

The spiral is not a burden. It is a reflection.

A correctly decomposed system — one where Engines contain logic and Accessors contain I/O and Managers contain orchestration and Utilities contain nothing domain-specific — produces a test suite that follows naturally from its structure. Unit tests are fast because Engines and Flows have no hidden dependencies. Integration tests are targeted because

Accessors have narrow, stable interfaces. E2E tests are confident because Experiences and Managers represent complete, semantically meaningful journeys. UAT aligns because those journeys were designed to represent actual human purpose.

The work of creating a good test suite is mostly the work of creating a good architecture. The spiral does not impose additional design constraints — it reads off the constraints already imposed by correct decomposition. When those constraints are satisfied, testing is an expression of the structure. When they are violated, testing is a fight against it.

Boundary-Driven Testing names that relationship precisely. Boundaries determine test scope. Boundaries define mock points. Boundaries generate test profiles. Fix the boundaries and the spiral follows.

✦ ✦ ✦

Appendix A: Glossary

Architecturally Significant Boundary — A seam between components with distinct roles, such as Engine-Accessor or Experience-Backend. The correct and stable placement for test doubles.

Boundary-Driven Testing — A test strategy in which architectural boundaries between component roles determine test scope, mock placement, and

assertion targets. Derived from VBD/EBD decomposition rather than imposed as an external testing framework.

Contract State — The set of possible return types a component can emit across its boundary, including success values, domain errors, and infrastructure failures. Tests must exercise every contract state to ensure the caller handles all outcomes.

Controllability — The ability to place a component into a known state without invoking the full system. Required for isolation.

End-to-End Test — A test that exercises a full user-visible journey through the real stack with no mocks. In BDT, E2E tests target Manager and Experience components to verify that assembled paths produce correct outcomes against live dependencies.

Integration Test — A test that verifies coordination between components at the Manager or Experience level, with dependencies mocked at the outer architectural boundary. In BDT, integration tests confirm that orchestration wiring matches the intended design.

Inverted Pyramid — A test suite dominated by E2E tests because unit and integration tests cannot exercise meaningful behavior. A structural indicator of misplaced boundaries, not a testing indicator.

Mock — A test double placed at an architecturally significant boundary to isolate the component under test from its dependencies. In BDT, mocks are positioned exclusively at role boundaries, not at arbitrary internal call sites.

Mock Proliferation — A test requiring many simultaneous mocks. A signal that the component crosses too many boundaries or has absorbed too many responsibilities.

Observability — The ability to determine what a component did given controlled inputs. Required for meaningful assertion.

Refactoring Confidence — The assurance that internal changes to a component are safe so long as its boundary tests continue to pass. BDT provides this by anchoring tests to stable architectural seams rather than internal implementation details.

Seam — The point between two component roles where a mock is placed during testing. Seams exist at architecturally significant boundaries and represent the natural isolation surface for test doubles.

Structural Signal — Testing difficulty — such as excessive mocking, brittle assertions, or unclear scope — that reveals architectural misalignment rather than a testing tooling problem. BDT treats these signals as prompts to fix boundaries, not tests.

System Test — A test that targets non-functional qualities such as resilience, performance, and deployment correctness. System tests use failure injection, load generation, and configuration variation against the full assembled system.

Test Profile — The characteristic test shape of a component role, specifying what is tested, what is mocked, and what is asserted. Each VBD/EBD role (Engine, Accessor, Manager, Utility, etc.) has a distinct test profile determined by its boundary relationships.

Test Spiral — A progression of test scope from unit through integration, end-to-end, system, and user acceptance. Each level corresponds to a level of architectural scope, and the spiral shape emerges naturally from correct boundary placement.

Unit Test — A test that exercises the internal logic of a single component — typically an Engine, Flow, or Utility — with all dependencies mocked at their architectural boundaries. In BDT, unit tests verify logic correctness and full contract-state coverage.

User Acceptance Test — A test performed by real users against the live system to verify that core journeys fulfill business intent. UATs use no mocks and validate that the system delivers actual value, not just technical correctness.

✦ ✦ ✦

Appendix B: BDT at a Glance

Spiral Level	Primary Structural Target	Mock Strategy	Assertion Scope
Unit	Engine · Flow · Utility · Interaction	Mock all dependencies through their contracts	Logic correctness; all contract states handled
Integration	Manager · Experience coordination	Mock at outer boundary; dependencies return controlled contract states	Collaboration wiring; orchestration matches design
End-to-End	Manager · Experience	No mocks; full stack; real external systems	User-visible outcome; journey completion
System	Full system	Failure injection; load; configuration variation	Non-functional qualities; resilience; deployment correctness
UAT	Core use case / Core user journey	None (real users)	Business intent fulfilled

✦ ✦ ✦

Appendix C: Case Study — E-Commerce Order Processing

This appendix presents a fictional but structurally realistic example of BDT applied to an e-commerce order processing system. The architecture follows VBD

role assignments. The test suite follows the spiral. The example demonstrates how boundary placement determines test scope, mock strategy, and maintenance cost — and how adding a new capability requires minimal test changes when the boundaries are correct.

C.1 System Architecture

The order processing system is decomposed into the following components:

Managers:

- **OrderManager** — orchestrates the order lifecycle: validates, prices, persists, and coordinates with peer managers for payment and notification. Contains no business logic. Sequences Engine calls and handles cross-cutting routing.
- **PaymentManager** — orchestrates payment processing workflows. Receives async requests from OrderManager, coordinates with payment-specific engines and accessors, and emits payment outcome events.
- **NotificationManager** — orchestrates customer notification delivery. Receives async events from OrderManager (and potentially PaymentManager), determines notification channels, and coordinates delivery.

Engines:

- **ValidationEngine** — applies order validation rules: required fields, item availability constraints, customer eligibility, promotion expiration. Reads from RulesResourceAccessor and InventoryResourceAccessor.
- **PricingEngine** — calculates order totals: base pricing, volume discounts, promotional codes, tax computation, currency handling. Reads from PricingResourceAccessor.

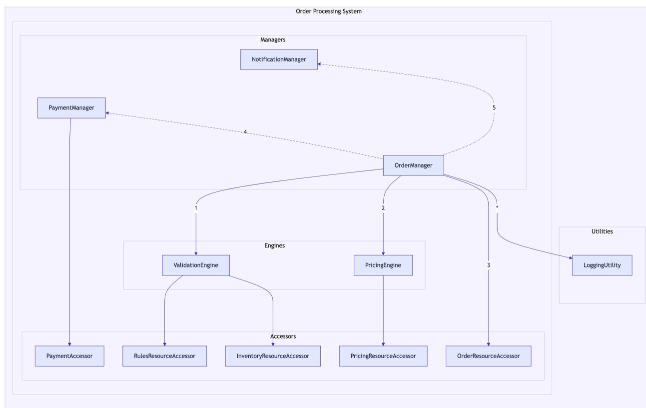
Resource Accessors:

- **OrderResourceAccessor** — translates domain order objects to and from the persistent store. Handles serialization, query construction, and connection management.
- **RulesResourceAccessor** — provides access to validation rules, business policies, and eligibility criteria.
- **InventoryResourceAccessor** — provides access to inventory levels, reservation state, and stock availability.
- **PricingResourceAccessor** — provides access to pricing tables, discount rules, tax rates, and promotional configurations.

Utilities:

- **LoggingUtility** — structured log emission. Pure sink: accepts log entries, formats them, writes to the configured transport. No domain awareness.

Note: OrderManager communicates with PaymentManager and NotificationManager via async messages. This is standard VBD inter-manager communication — managers do not call each other directly; they publish events that peer managers consume.



diagram

C.2 Unit Test Suite

Each Engine is tested in isolation. Accessors below each Engine are mocked. The tests exercise every branch of the Engine's logic by controlling the inputs and the mock responses.

ValidationEngine Unit Tests

Test Scenario	Input	Mock Setup	Expected Result
Missing required field (customer email)	Order with null email	None needed	`ValidationFailure("missing_required_field", "email")`
Empty line items	Order with zero items	None needed	`ValidationFailure("empty_order", null)`
Expired promotional code	Order with promo code "SUMMER2025"	RulesResourceAccessor returns promo with `expires: 2025-09-01`	`ValidationFailure("expired_promotion", "SUMMER2025")`
Out-of-stock item (validation-level check)	Order with item SKU-9912	InventoryResourceAccessor returns `stock_count: 0` for SKU-9912	`ValidationFailure("item_unavailable", "SKU-9912")`
Customer account suspended	Order with customer ID 4401	RulesResourceAccessor returns customer with `status: suspended`	`ValidationFailure("customer_ineligible", "account_suspended")`

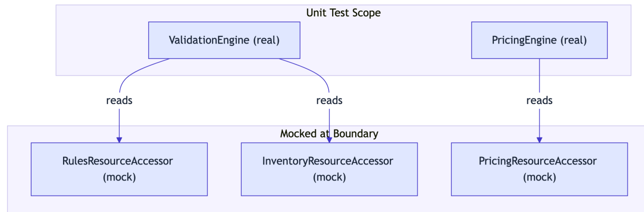
| Valid order, all checks pass | Complete order, all fields present | RulesResourceAccessor returns valid promo, active customer; InventoryResourceAccessor returns positive stock | `ValidationPass(order)` |

PricingEngine Unit Tests

Test Scenario	Input	Mock Setup	Expected Result
Volume discount threshold (100+ units)	Order with 150 units of SKU-1001	PricingResourceAccessor returns tier pricing: 100+ at 15% discount	`PricedOrder(discount_applied: "volume_15pct")`
Promotional code (percentage)	Order with promo "SAVE20"	PricingResourceAccessor returns promo: 20% off, no exclusions	`PricedOrder(promo_applied: "SAVE20", discount: 20%)`
Promotional code with exclusion	Order with promo "SAVE20", item in exclusion list	PricingResourceAccessor returns promo with exclusion list containing item SKU	`PricedOrder(promo_applied: null, exclusion_reason: "item_excluded")`
Tax calculation (multi-jurisdiction)	Order shipping to CA	PricingResourceAccessor returns CA tax rate 8.25%	`PricedOrder(tax_rate: 8.25%, tax_amount: computed)`

Combined volume + promo (stacking rules) | 150 units with promo "SAVE20" | PricingResourceAccessor returns stacking policy: "best_single" | `PricedOrder(discount_applied: "volume_15pct", stacking: "best_single_applied")` |
Zero-cost order (full discount) | Order fully covered by store credit | None needed | `PricedOrder(total: 0.00, payment_required: false)` |

Mock placement for all Engine unit tests:



diagram

C.3 Integration Test Suite

Integration tests exercise the seams between tiers. Each seam is tested with the caller real and the callee mocked, returning controlled contract states. The question is not whether the callee works — that is answered by its unit tests — but whether the caller handles every state the callee can emit.

Seam: OrderManager to ValidationEngine

Test Scenario | Mock Response from ValidationEngine | Expected Manager Behavior |

|—|—|—|

Validation passes | `ValidationPass(order)` | Manager proceeds to PricingEngine |

Validation fails (missing field) | `ValidationFailure("missing_required_field", "email")` | Manager returns rejection, no downstream calls made |

Validation fails mid-order (concurrent modification) | `ValidationFailure("order_modified_concurrently", order_id)` | Manager returns conflict error, logs warning, no payment attempted |

ValidationEngine throws unexpected error | `RuntimeException("database_unavailable")` | Manager returns system error, logs critical, no downstream calls |

Seam: OrderManager to PricingEngine

Test Scenario | Mock Response from PricingEngine | Expected Manager Behavior |

|—|—|—|

Pricing succeeds | `PricedOrder(total: 142.50)` | Manager persists order, publishes event to PaymentManager |

Pricing returns zero total | `PricedOrder(total: 0.00, payment_required: false)` | Manager skips PaymentManager, persists order, publishes to NotificationManager |

PricingEngine returns error | `PricingError("tax_service_unavailable")` | Manager returns pricing failure, no downstream calls |

Seam: OrderManager to PaymentManager (async)

Test Scenario | Mock Response from PaymentManager | Expected Manager Behavior |
|—|—|—|
Payment event accepted | `PaymentEventAccepted(correlation_id)` | Manager persists order as pending-payment, publishes to NotificationManager |
Payment queue unavailable | `QueueUnavailable("broker_down")` | Manager returns system error, logs critical, order not persisted |

Note: PaymentManager is a peer manager. OrderManager publishes payment requests asynchronously; payment outcomes arrive via separate event subscription. Integration tests verify message publication, not payment processing.

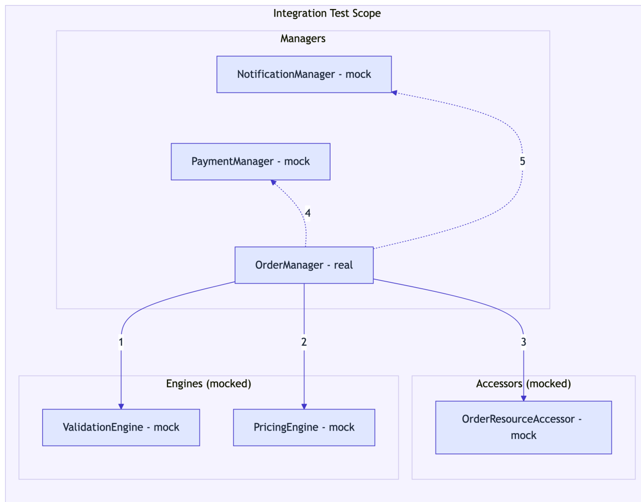
Seam: OrderManager to NotificationManager (async)

Test Scenario | Mock Response from NotificationManager | Expected Manager Behavior |
|—|—|—|
Notification event accepted | `NotificationEventAccepted(correlation_id)` | Manager completes order flow successfully |

Notification queue unavailable | `QueueUnavailable("broker_down")` | Manager logs warning, order still succeeds (notification is non-blocking) |

Note: NotificationManager is a peer manager. Notification failures do not block order completion — they are logged and retried independently.

Mock placement for integration tests:



diagram

C.4 End-to-End Test Scenarios

E2E tests exercise the full stack with no mocks. Real databases, real payment processing (sandbox mode), real event publication. The assertions target user-visible outcomes and verifiable side effects.

Scenario 1: Happy Path Order

1. Submit a valid order through the API with two line items and a promotional code.
2. Assert: API returns `201 Created` with an order confirmation containing the order ID, applied discount, and estimated delivery.
3. Assert: Order is retrievable via `GET /orders/{id}` with status `confirmed`.
4. Assert: Inventory counts for both items are decremented.
5. Assert: PaymentManager processes payment successfully (transaction appears in sandbox).
6. Assert: NotificationManager receives OrderConfirmed event asynchronously.

Scenario 2: Order with Payment Failure

1. Submit a valid order through the API using a test card number that triggers decline.
2. Assert: API returns `202 Accepted` (order submitted, payment pending).
3. Assert: PaymentManager emits PaymentFailed event.
4. Assert: Order is retrievable via `GET /orders/{id}` with status `payment_failed`.

5. Assert: NotificationManager receives PaymentFailed event and sends appropriate notification.

Scenario 3: Low Inventory Warning

1. Submit a valid order for 10 units where only 4 are in stock.
2. Assert: ValidationEngine detects low inventory via InventoryResourceAccessor.
3. Assert: API returns `201 Created` with order confirmation and low-stock warning.
4. Assert: Order persisted with inventory warning flag.
5. Assert: NotificationManager receives event with inventory details for customer communication.

C.5 Adding a New Payment Method

This section demonstrates the maintenance cost of adding a new capability — a cryptocurrency payment option — to the system. Because boundaries are correctly placed, the change is contained to a single new component.

Existing Architecture:

The system has one **PaymentAccessor** that provides multiple **PaymentProviders**. Each provider handles a specific payment method (credit card, PayPal, etc.) and implements the `PaymentProvider` contract. The PaymentAccessor uses dependency

injection to resolve the correct provider based on the payment method in the request. This design isolates payment-method-specific logic within providers while keeping the PaymentAccessor's orchestration stable.

```
PaymentManager
└── PaymentAccessor
├── CreditCardProvider (existing)
├── PayPalProvider (existing)
└── CryptoPaymentProvider (new)
```

What changes:

1. **CryptoPaymentProvider (new)** — A new provider to translate payment requests into cryptocurrency gateway API calls. Handles authentication, request formatting, response parsing, and error translation specific to the crypto provider. Implements the same `PaymentProvider` contract that existing providers implement. Registered with the DI container so PaymentAccessor can resolve it when `payment_method: "crypto"` is requested.
2. **One new Provider unit test** — Test that CryptoPaymentProvider correctly translates payment requests into the crypto gateway's API format and correctly parses responses. The mock

target is the crypto gateway's HTTP client. Contract states are identical to other payment providers: authorized, declined, timeout, error.

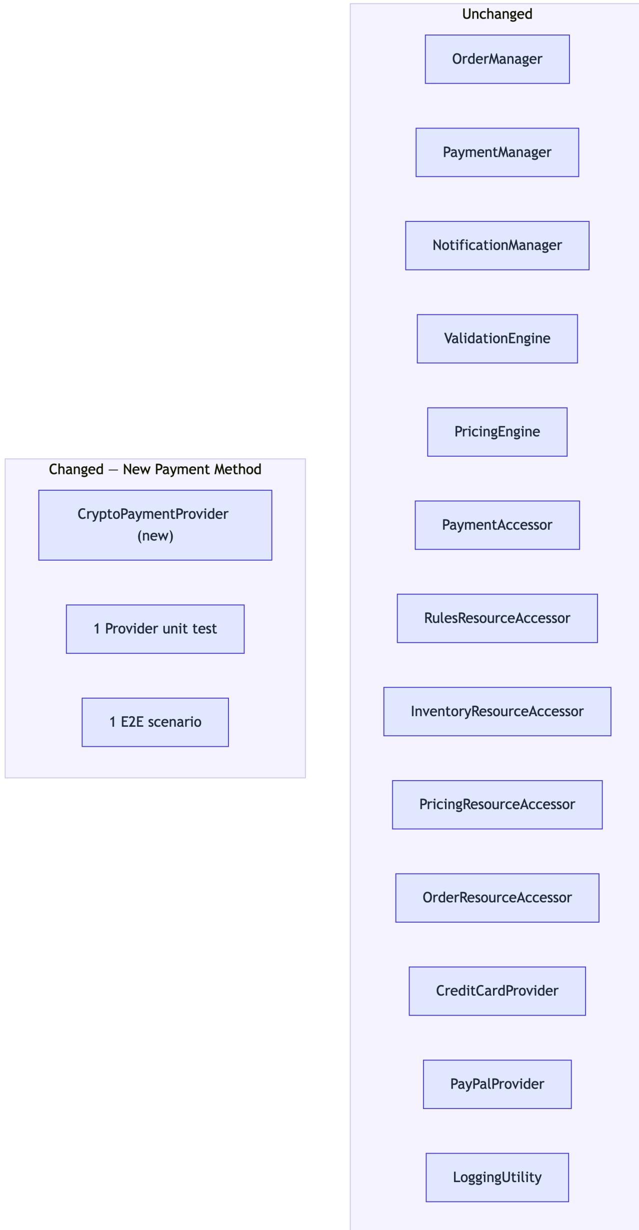
3. **One new E2E scenario** — Submit an order through the API with `payment_method: "crypto"`. Assert the same outcomes as the happy path: order confirmed, payment processed via crypto gateway, NotificationManager receives confirmation event.

What does not change:

- **PaymentAccessor** — uses dependency injection to resolve the correct provider based on payment method. No orchestration logic changes; it calls the same `PaymentProvider` contract regardless of which provider is injected.
- **PaymentManager** — delegates to PaymentAccessor; unaware of which provider handles the request.
- ValidationEngine unit tests — validation rules are payment-method-agnostic.
- PricingEngine unit tests — pricing is independent of payment method.
- OrderManager orchestration — it publishes to PaymentManager regardless of payment method.

- NotificationManager — it receives payment outcome events regardless of how payment was processed.
- All existing integration tests — PaymentAccessor's orchestration logic is unchanged.
- All existing E2E scenarios — the happy path and payment failure scenarios are unaffected.

Change impact diagram:


Changed — New Payment Method
CryptoPaymentProvider (new)
1 Provider unit test
1 E2E scenario
Unchanged
OrderManager
PaymentManager
NotificationManager
ValidationEngine
PricingEngine
PaymentAccessor
RulesResourceAccessor
InventoryResourceAccessor
PricingResourceAccessor
OrderResourceAccessor
CreditCardProvider
PayPalProvider
LoggingUtility

diagram

This is the structural payoff of correct boundary placement. The new payment method is volatile — it is a new external integration with its own protocol, authentication, and error semantics. But the volatility is contained entirely within the new Provider, which implements the existing `PaymentProvider` contract. PaymentAccessor does not change — it resolves the provider via dependency injection. PaymentManager does not change — it delegates to PaymentAccessor. OrderManager does not change — it still publishes payment requests to PaymentManager. NotificationManager does not change — it still receives payment outcome events. The Engines know nothing about payment methods. The test suite reflects this containment: two new tests for the new provider, zero modifications to existing tests, full confidence.

✦ ✦ ✦

References and Influences

William Christopher Anderson

Anderson, William Christopher. *Volatility-Based Decomposition in Software Architecture.* February 2026. `vbd.md`

Anderson, William Christopher. *Experience-Based Decomposition.* March 2026. `ebd.md`

VBD and EBD define the structural models and role taxonomies that BDT maps to the test spiral. The component roles (Manager, Engine, Resource Accessor, Utility; Experience, Flow, Interaction, Utility), communication rules, and core scenario validation mechanisms are taken from these sources. BDT establishes the testability consequences of those structures: where to test each role, what to mock, and how to read testing difficulty as structural signal.

David L. Parnas

Parnas, David L. "On the Criteria To Be Used in Decomposing Systems into Modules." *Communications of the ACM*, 1972.

Parnas argued that modules should hide design decisions likely to change behind stable interfaces. That same abstraction is what makes components testable: the stable interface is what callers depend on, and what tests can target without coupling to implementation. BDT extends this from the module level to the architectural level, applying the same principle across the full component role hierarchy.

Robert C. Martin

Martin, Robert C. *Clean Architecture.* Pearson, 2017.

Martin's work on dependency inversion, boundary placement, and the Dependency Rule establishes the structural conditions under which testing is tractable. His principle — mock across architecturally significant boundaries, but not within

them — is adopted directly in Section 4. His observation that testability is a primary benefit of correct dependency management is the foundational premise of BDT.

Martin Fowler

Fowler, Martin. "The Practical Test Pyramid." *martinfowler.com*, 2018.

Fowler's test pyramid established the proportional framing — many unit tests, fewer integration tests, fewer still E2E — and the practical consequences of inverting it. BDT builds on this by grounding the pyramid's levels in structural tiers rather than convention, explaining *why* the proportions emerge from correct decomposition rather than treating them as a design rule to follow.

Gregor Hohpe and Bobby Woolf

Hohpe, Gregor; Woolf, Bobby. *Enterprise Integration Patterns.* Addison-Wesley, 2003.

Hohpe and Woolf's treatment of integration points as volatility points reinforces the argument that Accessors are the natural integration test target. Their accessor and adapter patterns define the narrow, stable interfaces that integration tests verify — and that unit tests mock. The patterns also describe the specific behaviors (retries, error translation, protocol handling) that integration tests must exercise and unit tests must exclude.

Juval Löwy

Löwy, Juval. *Righting Software.* Addison-Wesley, 2019.

Löwy's IDesign methodology is the source of the Manager-Engine-Accessor taxonomy. The communication rules and role constraints that prevent runtime coupling also prevent test complexity — an observation that BDT makes explicit. The discipline that keeps Engines from coordinating workflows keeps unit tests focused. The discipline that keeps Accessors from applying business rules keeps integration tests targeted.

✦ ✦ ✦

Author's Note

Every test type described here — unit, integration, E2E, system, UAT — predates this paper by decades, and the relationship between architecture and testability is not a new observation. What the author has not encountered is a framework that makes this relationship explicit and structural: that derives the test profile of each component directly from its architectural role, that treats mock placement as boundary evidence, and that reads testing difficulty as a diagnostic signal about structural health rather than a tooling problem. Whether that specific synthesis

exists elsewhere under a different name, the author cannot say. What can be said is that the author has not seen it, despite having practiced and looked for it.

BDT is also inseparable from VBD and EBD — it is the natural consequence of decomposing correctly, made explicit. The intent is a reference suitable for engineering onboarding, test strategy discussions, and architectural review in organizations building products intended to last.

✦ ✦ ✦

Distribution Note

This document is provided for informational and educational purposes. It may be shared internally within organizations, used as a reference in testing and architecture discussions, or adapted for non-commercial educational use with appropriate attribution.

APPENDIX E

Project Design

Project Design

The Design of Projects: A Practitioner-Oriented Articulation

Author: William Christopher Anderson
Date: April 2026

Executive Summary

Projects fail not because teams lack effort, but because they lack informed planning. Traditional project management estimates cost and schedule from requirements alone, producing plans that are routinely two to four times off from reality. Project Design addresses this problem by treating the system architecture as the foundation for all planning decisions.

Rather than estimating from requirements documents or user stories, this approach derives the project plan directly from the architectural decomposition. Components become work packages. Dependencies between components become the project network. The critical path through that network determines the project duration. Compression and decompression of that path determine the feasible range of schedules. Risk is quantified objectively from the float distribution across all activities.

The result is not a single plan, but a set of viable options — typically three — spanning conservative, balanced, and aggressive approaches. Each option comes with quantified cost, schedule, risk, and staffing requirements. Management selects from these options in a structured decision meeting, committing resources or canceling the project based on objective trade-off analysis.

This methodology integrates established techniques from critical path analysis, earned value management, and risk quantification into a cohesive process that is compatible with both traditional and agile delivery frameworks. It provides architects, project managers, and engineering leaders with a practical, repeatable approach to planning software projects with confidence.

Abstract

Software project planning has historically operated independently of software architecture, resulting in plans that fail to account for the structural realities of the systems they propose to build. Estimation without decomposition produces unreliable forecasts. Scheduling without dependency analysis produces plans that collapse under execution pressure. Risk management without float analysis produces subjective assessments that fail to predict actual failures.

Project Design integrates system design with project planning by deriving the project plan directly from the architectural decomposition. This paper presents a structured methodology covering activity inventorying, network construction, critical path analysis, multi-level estimation, schedule compression, objective risk quantification, earned value validation, staffing principles, staged delivery, and structured decision-making. The approach produces multiple viable project options with quantified trade-offs, enabling rational decision-making by stakeholders.

The methodology is grounded in Juval Löwy's IDesign method and draws on established project management techniques, adapted for modern software development contexts including agile delivery. It is applicable to projects of any scale, from small team

efforts to large enterprise programs, and provides practitioners with a repeatable process for transforming architectural designs into executable project plans.

1. Introduction

Software projects exist in a state of chronic estimation failure. Industry data consistently shows that large software projects exceed their budgets by fifty to two hundred percent, miss deadlines by comparable margins, and deliver only a fraction of projected benefits. These failures are not primarily failures of execution — they are failures of planning.

The root cause is a fundamental disconnect between what is being planned and how the plan is constructed. Traditional project management treats software as a manufacturing process where effort can be estimated from requirements, parallelized across resources, and tracked against milestones defined by feature completion. This model fails because software is not manufactured — it is designed and integrated. The effort required to build a system depends not on the number of features, but on the structure of the system, the dependencies between its components, and the integration points where those components must work together.

Project Design addresses this disconnect by making the system architecture the foundation of the project plan. The premise is simple but profound: you cannot estimate what you have not designed. Without an architectural decomposition, you cannot identify the components that constitute the work. Without components, you cannot identify the dependencies that constrain sequencing. Without dependencies, you cannot identify the critical path that determines duration. Without duration, you cannot calculate cost. Without cost and risk, you cannot make informed decisions.

While the examples and terminology in this paper are drawn primarily from software engineering — where the methodology originated and where its application is most developed — the underlying techniques are not software-specific. Critical path analysis, float-based risk quantification, earned value tracking, compression analysis, and structured decision-making apply to any project with a decomposable structure and identifiable dependencies. In software, the architecture is the structure — the architectural decomposition provides the components, dependencies, and integration points from which the project plan is derived. In construction, product development, organizational transformation, or infrastructure programs, the

structural decomposition takes a different form, but the planning discipline is identical: design the project before you estimate it.

This paper presents a practitioner-oriented articulation of project design, covering the complete process from architectural review through structured decision-making. It describes how to construct project networks from architectural dependencies, analyze critical paths, estimate effort at multiple levels, compress schedules within feasible limits, quantify risk objectively, validate plans using earned value curves, staff projects according to architectural boundaries, and present options to stakeholders for informed decision-making.

2. Foundations

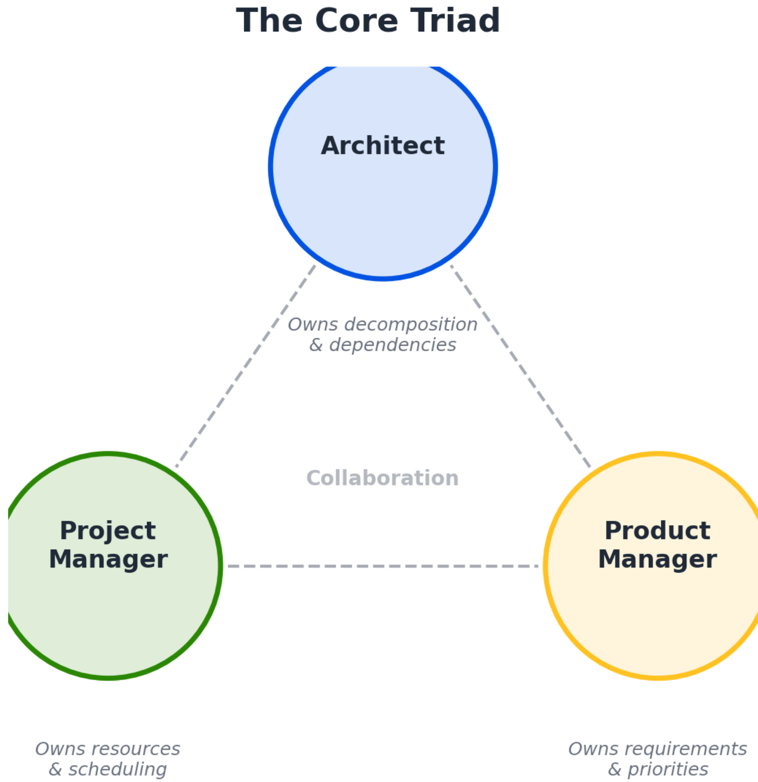
2.1 The Core Premise

Project design rests on a single foundational premise: the system architecture is the project plan. This is not a metaphor. The components identified during architectural decomposition become the work packages of the project. The dependencies between those components become the precedence relationships in the project network. The integration points where components must work together become the milestones of the project. The critical path through

those dependencies determines the duration. The staffing required to execute that path determines the cost.

This premise inverts the traditional relationship between architecture and project management. In conventional practice, project managers estimate effort from requirements, construct schedules from those estimates, and treat architecture as a technical detail handled during execution. In project design, the architecture comes first. It must be substantially complete before any meaningful estimation can begin. The investment in architectural design is not an overhead cost — it is the primary mechanism by which estimation accuracy is achieved.

2.2 The Core Triad



The Core Triad

Three roles collaborate throughout the project design process, each bringing a distinct perspective that the others cannot provide.

The Architect owns the system decomposition. They identify components, define dependencies, analyze the critical path, and provide technical risk assessment. They are responsible for the structural integrity of the plan.

The Project Manager owns resource allocation. They assess availability, cost, organizational constraints, and political feasibility. They translate the architectural plan into an executable staffing and scheduling strategy.

The Product Manager owns requirements prioritization. They serve as the customer proxy, resolving requirement conflicts, defining priorities, and managing stakeholder expectations.

In agile contexts, these roles map to the Tech Lead, Scrum Master or Delivery Lead, and Product Owner respectively. The important principle is that all three perspectives are present throughout the planning process. Removing any one of them produces plans with blind spots that manifest as failures during execution.

2.3 The Fuzzy Front End

The period between project inception and the commitment to a specific plan is the fuzzy front end. During this period, the core triad works to produce the architecture, the project design, and the options for management. This period typically consumes fifteen to twenty-five percent of the total project duration.

This investment is not wasted time. It is the mechanism by which the remaining seventy-five to eighty-five percent of the project is executed efficiently. Projects that skip or compress the front end invariably spend more total time and money than those that invest in it, because the cost of planning failures during execution far exceeds the cost of planning itself.

Staffing during the front end is minimal — only the core triad. Full team staffing begins after management commits to a plan. This means the project's most expensive resources — the full development team — are not engaged until the plan is validated, reducing financial exposure.

3. Activity Inventory

The first step in project design is constructing a complete inventory of all activities required to deliver the system. This inventory must be comprehensive — the most common source of estimation error is not incorrect estimates for known activities, but the complete omission of activities that were never identified.

3.1 Architecture-Derived Activities

The architectural decomposition provides the primary source of activities. Each component identified during design becomes a work package with a predictable lifecycle: detailed design, implementation, unit testing,

code review, and documentation. This covers both backend components — Manager, Engine, Resource Accessor, Utility — and frontend components — Experience, Flow, Interaction, Utility — when the system includes a user interface layer decomposed using Experience-Based Decomposition. Each pair of connected components generates an integration activity. Each use case or core user journey generates a system-level verification activity.

3.2 Non-Code Activities

A software project involves substantially more than writing code. The activity inventory must explicitly include activities that are routinely underestimated or omitted entirely. These fall into several categories.

Planning and design activities include requirements analysis and refinement, architectural design, project design itself, technology spikes and prototyping, and vertical slice or proof-of-concept development.

Organizational activities include stakeholder communication and alignment, change management and organizational readiness, vendor coordination and procurement, licensing, and legal or regulatory review where applicable.

Quality and compliance activities include security review and compliance audit, accessibility audit, user acceptance testing, performance and load testing, security penetration testing, and disaster recovery testing.

Operational activities include environment setup for development, staging, and production; continuous integration and deployment pipeline setup; monitoring, alerting, and observability infrastructure; runbook and playbook creation; on-call training and rotation setup; release planning and communication; go-live and cutover planning; and rollback planning.

People and knowledge activities include learning curves for new technology or domain knowledge, training and knowledge transfer, documentation of architecture, APIs, and user-facing content, and team onboarding materials.

Post-launch activities include a hypercare or stabilization period, post-mortem or retrospective sessions, and a warranty support period.

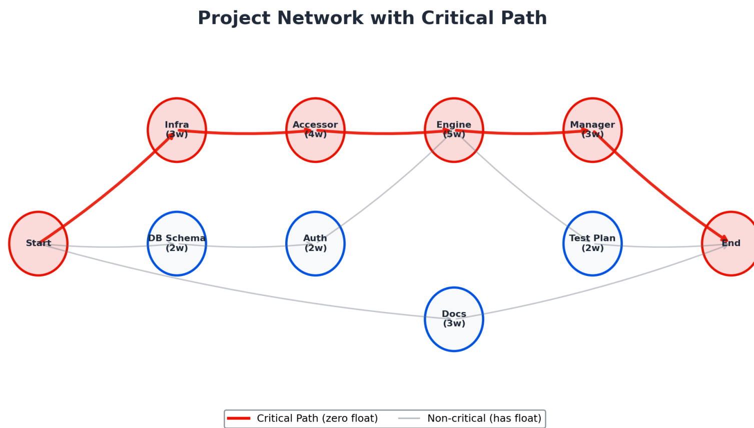
The distinction between code activities and non-code activities is critical because non-code activities often account for thirty to fifty percent of total project effort, yet are routinely excluded from initial estimates.

3.3 The Activity Lifecycle

Each component follows a lifecycle that should be estimated as a unit. A common rule of thumb suggests that total effort is on the order of six times the design

effort: roughly one unit for design, three for implementation, one for unit testing, and one for integration testing. This ratio varies significantly by complexity, domain, and team experience, but provides a useful starting point for calibration.

4. Network Construction



Project Network with Critical Path

With the activity inventory complete, the next step is constructing the project network — a directed graph that models the precedence relationships between all activities. The network is the structural foundation for all subsequent analysis.

4.1 Deriving Dependencies from Architecture

The architectural decomposition determines the dependency ordering. For backend systems decomposed using VBD: infrastructure and utilities have no upstream dependencies and form the leaf nodes of the network; Resource Accessors depend on infrastructure; Engines depend on Resource Accessors; Managers depend on Engines and Resource Accessors. For frontend systems decomposed using EBD: Utility components have no upstream dependencies; Interaction components depend on Utilities; Flow components depend on their constituent Interactions; Experience components depend on their constituent Flows. Frontend Experiences depend on backend Managers at the API boundary. This combined ordering provides the skeleton of the project network.

Non-code activities are inserted into the network at appropriate positions. Preparation activities such as requirements and architecture precede all construction. Test planning runs parallel to early construction. Integration testing follows the completion of connected component pairs. System testing follows the completion of all Managers. Documentation and training activities may run parallel to late-stage construction.

4.2 Dependency Consolidation

Unconsolidated networks contain redundant dependencies that distort analysis. If activity A depends on B and B depends on C, then A implicitly depends on C and the explicit dependency should be removed. Failing to consolidate dependencies inflates the apparent complexity of the network, produces incorrect float values, and obscures the true critical path.

A well-consolidated network has three to four dependencies per activity. Networks with higher dependency counts should be evaluated for architectural restructuring. If the project network is too complex, the architecture may need to be simplified.

4.3 Milestone Placement

Infrastructure completion should be pushed to milestone nodes that separate preparation from construction. This reduces dependency fan-out and simplifies the network. The project should have a single start event and avoid multiple end points.

4.4 Network Representation

Two representations are common. Node diagrams place activities on nodes with arrows representing dependencies. Arrow diagrams place activities on arrows with nodes representing events or milestones. Arrow diagrams, while less intuitive initially, provide

superior visualization for project design because they can be drawn to scale, with arrow length proportional to duration, making the schedule visible at a glance.

5. Critical Path Analysis

The critical path is the longest path through the project network from start to finish. Its duration is the duration of the project. No project can be accelerated beyond its critical path regardless of resource availability.

5.1 Forward and Backward Pass

The forward pass calculates the earliest possible start and finish time for each activity by traversing the network from start to finish. The earliest start of each activity is the maximum of the earliest finish times of all its predecessors.

The backward pass calculates the latest allowable start and finish time for each activity by traversing the network from finish to start. The latest finish of each activity is the minimum of the latest start times of all its successors.

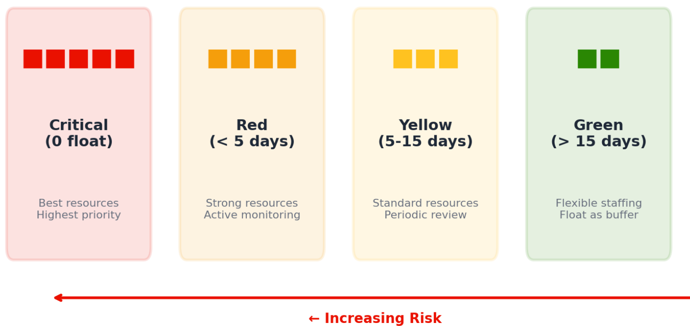
5.2 Float Analysis

The difference between the latest and earliest start times of an activity is its total float — the amount of time the activity can slip without delaying the project. Activities with zero total float are on the critical path.

Free float measures how much an activity can slip without delaying any immediate successor. Interfering float is the difference between total float and free float — consuming interfering float delays successors but not the project.

Float is the objective measure of risk. Activities with zero float are maximally risky — any delay directly extends the project. Activities with large float can absorb delays without consequence. The distribution of float across all activities determines the overall risk profile of the project.

5.3 Float Classification



Float Classification Drives Resource Assignment

Activities are classified by their float into risk categories. Critical activities have zero float and receive the highest resource priority. Red activities

have small float — typically less than five days — and are near-critical. Yellow activities have moderate float and represent medium risk. Green activities have large float and can be staffed flexibly.

This classification directly drives resource assignment. The best resources are assigned to the critical path. Strong resources go to near-critical activities. Float on non-critical activities can be traded for resource flexibility.

5.4 Proactive Risk Management

The primary reason well-managed projects slip is that non-critical activities consume their float and become critical, creating a new critical path that was not planned for and does not have the best resources assigned to it. Continuous monitoring of float degradation on near-critical chains enables proactive resource reassignment before this transition occurs.

6. Estimation

Estimation in project design operates at three levels, each serving a distinct purpose. The levels validate each other — significant discrepancies between levels indicate problems that must be resolved before proceeding.

6.1 Activity-Based Estimation

The primary estimation method is bottom-up, activity-by-activity estimation. Each activity is estimated individually, ideally by the person who will perform

it. Estimates are expressed in units of five days, aligning with week boundaries and reducing waste in the plan.

The estimate for each activity must include its complete lifecycle — not just implementation, but design, testing, review, and documentation. Activities that are routinely omitted from estimates include learning curves for new technology, test harness construction, deployment and installation effort, integration with adjacent components, and documentation.

Both underestimation and overestimation are harmful. Underestimation produces schedule pressure, corner-cutting, and quality degradation. Overestimation produces gold plating, scope creep, and inverted priority structures where padding becomes an entitlement. The goal is nominal estimation — the most likely duration assuming normal conditions and competent execution.

6.2 Broadband Estimation

Broadband estimation gathers input from twelve to thirty diverse participants through successive refinement iterations. The statistical advantage of group estimation is that the error of the sum is less than the sum of the errors — individual overestimates and underestimates tend to cancel.

The participant pool should include a diverse profile: veterans and newcomers, specialists and generalists, optimists and skeptics. In agile contexts, Planning Poker is a form of broadband estimation.

Broadband estimation serves as a sanity check against the detailed bottom-up estimate. It is not actionable on its own — it lacks the structural precision needed for network analysis. But significant discrepancies between broadband and detailed estimates indicate areas requiring investigation.

6.3 Overall Project Estimation

Historical data and estimation tools provide a third validation point. Comparison of the bottom-up estimate against historical performance on similar projects reveals systemic biases — teams that consistently underestimate integration effort, for example, or that fail to account for organizational overhead.

6.4 Cost Calculation

Project cost is the area under the staffing curve — the integral of staffing level over time. This calculation requires both the staffing distribution and the cost rates for each role. Cost consists of direct costs — developer salaries, equipment, tools, and licenses — which scale with team size and duration, and indirect costs — management overhead, facilities, and administrative support — which scale primarily with duration.

The total cost is a quadratic polynomial of time, combining the linear indirect costs with the nonlinear direct costs. This relationship means that both excessively compressed and excessively extended projects are more expensive than the optimal point, which typically falls near the all-normal solution.

6.5 Efficiency

Project efficiency measures the ratio of productive effort to total effort. For complex projects with multiple contributors, efficiency in the range of fifteen to twenty-five percent is common — the remainder is consumed by communication overhead, coordination, waiting, and context switching. An efficiency calculation significantly exceeding this range may indicate that dependencies and communication overhead have been underestimated.

7. Schedule Compression

Schedule compression reduces the project duration below the all-normal solution. There are exactly two mechanisms for compressing a schedule: making individual activities shorter, and restructuring the network to enable more parallelism.

7.1 Activity Compression

A top-performing resource can often complete an activity in less time than the normal estimate, but at a disproportionately higher cost — the relationship between time saved and cost incurred is not linear.

The exact ratios depend on domain, team composition, and activity type, and should be calibrated from historical data where available. The underlying principle is consistent: compression trades cost for time, and only activities on or near the critical path should be compressed — compressing non-critical activities does not accelerate the project.

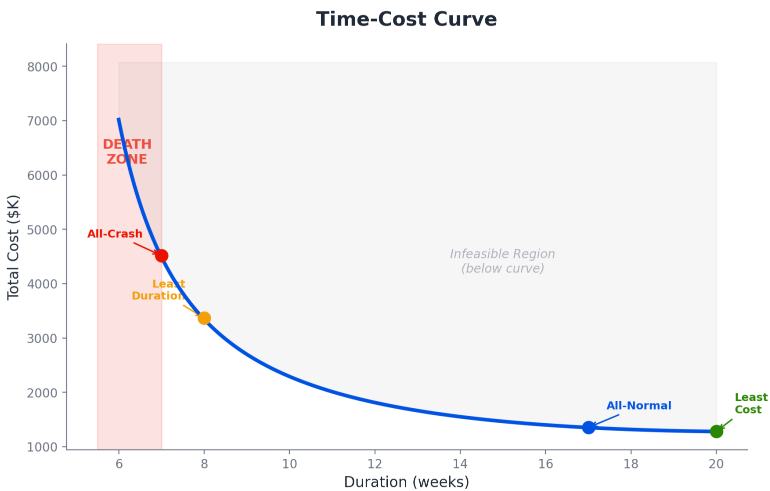
Compression has diminishing returns. Each activity compressed on the critical path may create a new critical path through activities that were previously non-critical. These new critical path activities are typically staffed with weaker resources, creating fragility. As a general guideline, the practical maximum compression tends to fall in the range of twenty-five to thirty percent of the normal duration, though this varies by project structure and resource availability.

7.2 Network Restructuring

The more profound acceleration technique is restructuring the network to enable parallel work. This can be achieved by investing in detailed design and contracts that allow implementation against interfaces rather than implementations, developing simulators that decouple Manager development from Engine and Accessor availability, and separating design activities from construction activities to enable pipelining.

Network restructuring requires additional investment — simulators must be built, contracts must be specified in detail, and coordination overhead increases. The cost is higher, but the schedule reduction can exceed what activity compression alone can achieve.

7.3 The Time-Cost Curve



Time-Cost Curve

Plotting all feasible solutions — from maximum compression to padded decompression — produces the time-cost curve. This curve is fundamental to project design. Points below the curve are infeasible — they cannot be achieved regardless of resources. Points on the curve represent efficient solutions. Points above the curve represent wasteful solutions.

The curve has several named points. The all-normal point represents all activities at nominal duration. The least-cost point is typically near the all-normal point and represents the minimum total cost. The all-crash point represents maximum compression of all critical activities. The least-duration point represents the absolute minimum achievable schedule, including network restructuring.

The region to the left of the least-duration point is the death zone — no amount of resources can achieve these schedules. When management requests a delivery date in the death zone, the project designer's responsibility is to communicate this clearly with supporting analysis.

7.4 Compression and Complexity

The true cost of compression is not financial — it is complexity. Project execution complexity, measured as the cyclomatic complexity of the project network, increases nonlinearly with compression. A normal solution might have a complexity of five to eight. A compressed solution might reach ten to twelve. The practical maximum complexity is ten to twelve, beyond which project management becomes unmanageable. A fully parallel project has complexity equal to the number of activities, which is always impractical.

8. Risk Quantification

Risk in project design is measured objectively from the float distribution. Risk represents the fragility of the plan — its sensitivity to activities slipping — not the probability of specific events occurring.

8.1 Criticality Risk Index

The criticality risk index classifies all activities by their float into four categories — critical, red, yellow, and green — and computes a weighted average. The formula assigns the highest weight to critical activities and the lowest to green activities, producing a value between 0.25 and 1.0.

The risk interpretation scale provides guidance for evaluating the result. Values below 0.30 indicate an over-decompressed project — resources are being wasted on excessive buffer. Values between 0.30 and 0.50 represent the comfortable zone. The design target is approximately 0.50, which balances schedule efficiency against fragility. Values between 0.50 and 0.75 indicate increasing fragility. Values above 0.75 should be avoided — the plan is too brittle to survive normal execution variability.

Risk is not linear. A risk index of 0.69 is substantially more dangerous than 0.50 — the relationship is more analogous to the Richter scale than to a linear percentage.

8.2 Activity Risk Index

The activity risk index uses actual float values rather than categorical classifications, providing finer-grained measurement. It is computed as one minus the normalized sum of float values across all activities. This measure is sensitive to outlier activities with extremely large float values, which can be addressed by capping outliers at one standard deviation above the mean.

8.3 Risk Decompression

Deliberately planning for a later delivery date than the normal solution reduces risk by adding float to near-critical activities. Even small decompression — two weeks beyond the normal solution — can significantly reduce the risk index. The optimal decompression point is typically found at the second derivative zero of the risk curve, which produces a risk index of approximately 0.50.

The cost of decompression is the additional direct cost of the extended schedule. In practice, this cost is modest compared to the risk reduction achieved, making decompression one of the most cost-effective risk mitigation strategies available.

8.4 Risk Cross-Over Point

The risk cross-over point is the schedule duration at which risk begins rising faster than cost. Beyond this point, further compression becomes unwise because the fragility of the plan increases disproportionately

relative to the cost savings achieved by the shorter schedule. The cross-over analysis produces an acceptable risk zone bounded by upper and lower cross-over points.

8.5 Risk During Execution

Once a plan is selected and execution begins, risk should be tracked continuously. A well-executed project shows a gradual, smooth decline in risk as activities complete and float is consumed predictably. A sudden jump in measured risk — a discontinuity in the risk curve — indicates a pending problem and warrants immediate investigation.

9. Options and Decisions

The culmination of project design is the presentation of options to management. This is not a status report or a request for permission — it is a structured decision framework that enables rational choice among viable alternatives.

9.1 The Three Options

At least three project design options should be presented, spanning the feasible zone of the time-cost curve.

The conservative option offers the longest schedule, lowest cost, and lowest risk. It uses minimal compression, provides substantial float buffers, and accommodates uncertainty in requirements, technology, and team capability.

The balanced option represents the recommended approach for most projects. It applies moderate compression to the critical path, achieves a risk index near the decompression target of 0.50, and balances schedule efficiency against plan fragility.

The aggressive option offers the shortest feasible schedule at the highest cost and risk. It applies maximum practical compression, requires top resources on all critical activities, and provides minimal buffer for absorbing delays.

Each option must include quantified values for duration, direct and indirect cost, risk index, peak staffing, critical path identification, milestone schedule, key assumptions, and the top three to five specific risks.

9.2 The Feed Me or Kill Me Decision

Management reviews the options and makes one of two decisions: commit resources to the selected option, or cancel the project. Both outcomes are valid. Killing a project that cannot be executed within acceptable cost, schedule, and risk parameters is a responsible act that preserves resources for better opportunities.

The architect recommends. Management decides. The architect's responsibility is to illuminate the trade-off space with objective analysis. Management's responsibility is to navigate that space based on business priorities, strategic considerations, and risk tolerance.

Never staff up before this decision. Never let management select an option outside the feasible zone. Never present a single plan — that is a demand, not a recommendation.

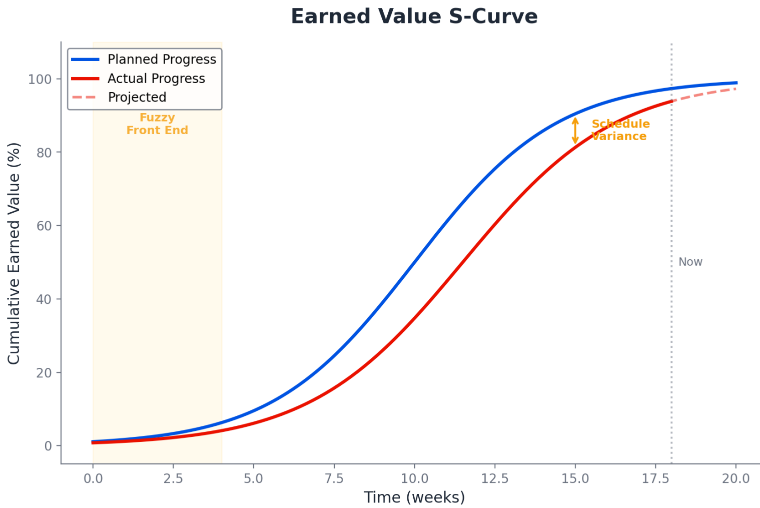
9.3 Post-Decision

Once management selects an option, several commitments follow. Scope is locked and changes go through formal change control. Resources are assigned according to the staffing plan for the selected option. Milestones are set. Staged delivery begins with the first stage — typically infrastructure and preparation. Earned value tracking commences against the selected option's planned progress curve.

10. Earned Value Planning and Tracking

Earned value serves dual purposes in project design: as a validation tool during planning and as a tracking tool during execution.

10.1 Plan Validation



Earned Value S-Curve

Each activity is assigned a value representing its contribution to system completion as a percentage of total project work. Plotting cumulative earned value over time for a well-designed plan produces a shallow S-curve — slow initial progress during the core team's front-end work, accelerating progress during the ramp-up and construction phases, and decelerating progress during integration and testing.

The shape of the S-curve validates the plan's sanity. A steep early curve indicates unrealistic optimism. A flat early curve with a steep late section indicates unrealistic pessimism. A straight line indicates fixed team size, which is nearly always

suboptimal. A very shallow, nearly straight curve indicates sub-critical staffing. A well-designed plan produces an S-curve with a coefficient of determination exceeding 0.95 when fit to a third-degree polynomial.

10.2 Execution Tracking

During execution, three lines are tracked: the planned progress curve, the actual progress curve, and the actual effort curve. The relationship between these curves provides diagnostic information.

When progress tracks the plan and effort tracks the plan, the project is healthy. When progress falls behind while effort exceeds the plan, the project is underestimating complexity. When progress leads while effort is below the plan, the project may be overestimating or the team may be sandbagging. When progress falls behind while effort dramatically exceeds the plan, there is a resource leak — effort is being consumed without proportional progress.

The critical capability of earned value tracking is projection. By extrapolating actual progress and effort curves, the project manager can project the completion date and final cost while there is still time to take corrective action. This forward-driving analysis is the essence of project management — the ability to project is the essence of a project.

11. Staffing

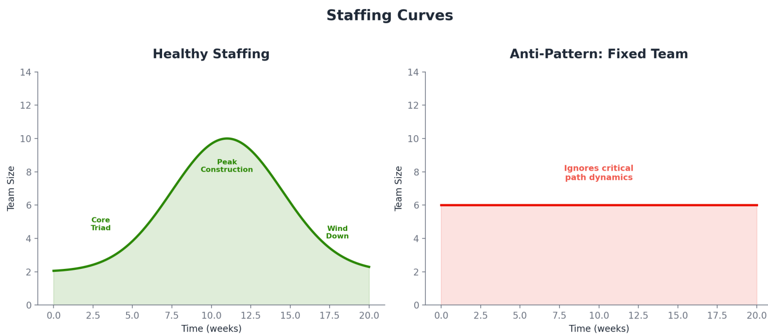
Staffing in project design follows directly from the architectural decomposition and the selected project option.

11.1 The One-to-One Rule

Each component is assigned to one developer. This principle enables reliable estimation, clear ownership, and minimal communication overhead. Assigning multiple developers to a single component introduces coordination costs that are difficult to estimate and manage. If a component cannot be built by one developer within the project timeline, it should be decomposed further — this is an architectural problem, not a staffing problem.

This principle is a direct application of Conway's Law: the interaction between team members mirrors the interaction between the components they build. A well-decomposed architecture with minimized inter-component coupling naturally minimizes inter-developer communication overhead.

11.2 The Staffing Curve



Staffing Curves

A properly planned project produces a staffing curve shaped like a smooth hump: a small core team during the front end, a gradual ramp-up as construction begins, a peak during maximum parallel activity, and a gradual wind-down during integration and testing.

Staffing anti-patterns include peaks and valleys, which indicate poor float utilization; erratic patterns where team members join, leave, and rejoin, which is organizationally impractical; steep ramps, which exceed teams' absorption capacity; and fixed team sizes, which ignore critical path dynamics.

Developers cannot come and go elastically. Staffing changes have ramp-up costs, context-switching overhead, and organizational friction. The staffing curve must be realistic — smooth transitions that real organizations can execute.

11.3 The Hand-Off Point

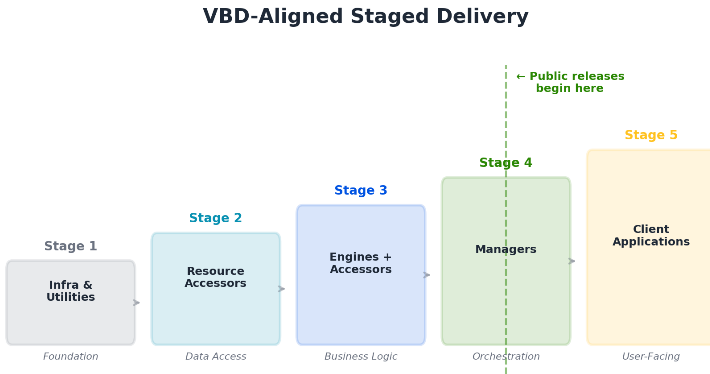
The hand-off point is where the architect transfers design responsibility to developers. At the senior hand-off point, the architect specifies service-level contracts and interfaces, and developers handle detailed design and implementation. This is faster, cheaper, and enables pipelining of architect design with developer construction, but requires senior developers.

At the junior hand-off point, the architect specifies detailed class-level design, and developers implement to specification. This is disproportionately more work for the architect but necessary when the team lacks design experience. The hand-off point must match the actual team composition — using a senior hand-off point with a junior team produces architectural drift and quality problems.

12. Staged Delivery and Integration

Project design always uses staged delivery. The system is delivered in incremental stages, each producing a working increment that demonstrates progress and enables early feedback.

12.1 VBD-Aligned Stages



VBD-Aligned Staged Delivery

In systems built using Volatility-Based Decomposition, stages align naturally with the component taxonomy. The first stage delivers infrastructure and utilities — the foundation on which everything else builds. The second stage delivers Resource Accessors — verified data access and external integration. The third stage delivers Engines with their Accessors — working business logic. The fourth stage delivers Managers — complete workflows and orchestration. The fifth stage delivers client applications and user interfaces.

Public releases begin after Managers are complete, because only at that point do user-visible workflows function end-to-end. Earlier stages are internal releases that build confidence and enable early integration testing.

12.2 Integration Planning

The necessary and sufficient unit of integration is two connected services. Integration begins bottom-up from leaf nodes and proceeds upward through the dependency graph. Each integration point is small, focused, and localizes bugs to the newest component added.

Milestones are based on integration, not features. Features are aspects of integration, not implementation — a feature only exists when the components required to produce it are integrated and functioning together. Planning around feature completion crosses component boundaries, obscures ownership, and produces milestones that are difficult to verify objectively.

13. Special Situations

13.1 God Activities

God activities are activities that are large both in absolute duration and relative to the mean activity size — typically more than one standard deviation above the mean. They are almost always on the critical path

and deform all project design techniques: risk models yield misleadingly low values, float analysis is skewed, and compression analysis becomes unreliable.

God activities should be addressed before trusting any analysis built on the network. Solutions include splitting the activity into internal phases that can overlap, developing simulators to reduce the blocking effect on dependent activities, and factoring the activity into a separate sub-project.

13.2 Very Large Projects

Projects with hundreds or thousands of activities exceed the human capacity for network comprehension, which is approximately one hundred activities. Research by Bent Flyvbjerg demonstrates that project size maps directly to poor outcomes — one in six medium-size projects experiences two hundred percent cost overrun and seventy percent schedule overrun.

The solution is the network of networks approach: decompose the large project into manageable sub-projects, each with its own critical path, risk analysis, and staffing plan. A preliminary mini-project discovers the network of networks structure. The primary investment for decoupling sub-networks is architecture and interface definition — the same discipline that enables good system decomposition also enables good project decomposition.

13.3 Small Projects

Small projects are paradoxically more sensitive to project design mistakes than large ones. With small teams on short durations, the resolution of any planning error is substantial relative to the total timeline. Almost every mistake becomes critical. Project design should be applied when the team size exceeds the number of network paths — which for small projects is often the case.

13.4 Sub-Critical Staffing

When resource availability constrains the project below the minimum staffing needed for the critical path, the project enters the sub-critical zone. Sub-critical staffing tends to increase cost substantially — often by twenty-five percent or more — while extending the schedule by a comparable margin and pushing the risk index into dangerous territory. There are no savings from sub-critical staffing — the combination of extended duration and increased overhead consistently exceeds the cost of adequate staffing.

The telltale sign of sub-critical staffing is an earned value curve that approximates a straight line rather than an S-curve. The staffing distribution is anemic — truncated and flat, missing the healthy ramp-up hump. In the extreme case of a single developer, all activities are serial, all are critical, risk is 1.0, and duration equals the sum of all effort.

14. Alignment with Agile Methodologies

Project design is not opposed to agile delivery — it is complementary to it. The distinction is between planning and execution. Project design provides the planning discipline that determines what is built, in what order, with what resources, and at what cost. Agile methods provide the execution discipline that governs how the work is performed day to day.

14.1 Agile Mappings

The Feed Me or Kill Me decision maps to Sprint Zero or PI Planning commitment. The core triad maps to the Product Owner, Scrum Master, and Tech Lead. The activity inventory maps to the refined backlog with architectural dependencies. The three options map to release plan scenarios. Broadband estimation maps to Planning Poker. Sprint velocity maps to the earned value rate. The S-curve maps to the cumulative flow diagram or burnup chart.

Staged delivery maps to Program Increments or release trains. Internal stages map to sprints with internal demos. Public releases map to production deployments.

14.2 Good Agile, Bad Agile

Good agile combines architectural discipline with iterative delivery — an effective, proven approach. Bad agile uses agile terminology as a justification for skipping architecture and project design, producing teams that are busy but not productive. The

distinguishing characteristic is whether the team has a structural understanding of what they are building and a quantified plan for how they will build it.

All agile construction techniques — stand-ups, Kanban, user stories, burndown charts — are assembly techniques. They are excellent for execution. But they are not design techniques. Just as lean manufacturing depends on meticulous design of both the product and the assembly line, good agile depends on meticulous architecture and project design. Features are aspects of integration, not implementation — plucking stories off a Kanban board and coding without structural understanding is the epitome of bad agile.

15. Project Recovery

When a project has failed its plan — missing every deadline, exceeding every budget, degrading in quality to the point where fixes create more defects than they resolve — recovery is the appropriate intervention. Recovery is rescue, not rehabilitation. It is short-term, decisive, and often uncomfortable.

Recovery begins with an honest assessment of whether the project can be saved. Not all projects can. The assessment examines root causes rather than symptoms — date, budget, and quality are symptoms, not causes. Causes are typically deterministic failures

in project inception, execution failures during construction, or process failures in tracking and response.

A successful recovery requires a recovery lead with the authority to make unquestioned demands, the trust of senior executives, and the willingness to make decisive changes. The recovery project design is short-term, not aspiring for fastest or cheapest, designed around 0.50 risk, and meticulous. The first one to two milestones must be delivered on schedule and on budget to rebuild trust.

The most common recovery mistakes are throwing more people at the problem, working harder doing more of the same, and the defibrillator effect — weak recovery attempts that produce a temporary improvement followed by relapse.

16. The Meta-Process

Project design itself is a project with its own activities, dependencies, and critical path. Understanding this meta-process helps plan the planning effort.

The meta-process proceeds through four phases. The preparation phase covers use case and call chain identification, architectural decomposition, non-code activity identification, and estimation at all three levels. The solutions exploration phase covers the normal solution, limited resource scenarios, sub-critical analysis, and progressive compression using

top resources, parallel work, activity changes, and full crashing. The analysis phase covers throughput and efficiency analysis, time-cost curve construction, risk decompression, risk quantification and modeling, and exclusion zone identification. The decision phase covers option recommendation, staged delivery planning, milestone identification, and the Feed Me or Kill Me review.

A seasoned architect can complete a single planning option in a few hours. A full project design with multiple options and complete analysis takes a few days to a week. The return on this investment is substantial — even a few percentage points of improved estimation accuracy on a large project justifies the effort many times over.

17. Conclusion

Project design transforms project planning from a subjective exercise in estimation and hope into an objective, structured process grounded in the realities of the system being built. By deriving the project plan from the architectural decomposition, this approach ensures that the plan reflects the actual work, dependencies, and risks of the project rather than abstract estimates disconnected from structural reality.

The methodology is not complex. Construct the network from the architecture. Find the critical path. Estimate activities. Compress within feasible limits.

Quantify risk from floats. Validate with earned value curves. Present options to management. Let them decide.

What makes this approach effective is not sophistication but discipline — the discipline to design before estimating, to quantify before deciding, and to present options rather than demands. Projects that apply this discipline consistently deliver more predictably than those that do not, not because the methodology eliminates uncertainty, but because it makes uncertainty visible and manageable.

The most dangerous project is not the one with high risk — it is the one where risk is unknown. Project design ensures that risk is known, quantified, and communicated. From that foundation, good decisions follow.

"Where there is no vision, the people perish." — Proverbs 29:18

Appendix A: Glossary

Activity Network — The directed graph of work packages and their dependencies that defines the execution order and parallelism of a project. It is derived from the architectural decomposition, not from team preferences.

All-Crash Solution — The schedule produced when every activity is compressed to its minimum possible duration using maximum resources. It represents the shortest feasible project duration and the highest cost point on the time-cost curve.

All-Normal Solution — The schedule produced when every activity uses its normal (uncompressed) duration. It represents the lowest-cost feasible schedule and serves as the starting point for compression analysis.

Boundary Work Package — A work package located at an integration point with external systems or other teams. Boundary work packages carry interface risk but tend to have lower estimation uncertainty than core work packages.

Compression — Reducing an activity's duration by adding resources, typically at increasing marginal cost. Compression is only applied to critical-path activities, and only when the cost-per-unit-time saved is justified.

Core Work Package — A work package containing execution logic — the algorithmic or business-rule heart of a component. Core work packages carry the highest estimation uncertainty in the network.

Critical Path — The longest sequence of dependent activities from project start to finish; determines minimum project duration.

Decompression — Deliberately extending the project schedule beyond the all-normal solution to reduce risk.

Earned Value — A metric tracking the value of completed work against the planned value at each point in time.

Feed Me or Kill Me — The decision point where management commits resources to a selected project option or cancels the project.

Float (Slack) — The amount of time an activity can slip without delaying the project (total float) or its immediate successors (free float).

God Activity — An activity that is disproportionately large relative to other activities, deforming project analysis techniques.

Hand-Off Point — The level of design detail at which the architect transfers work to developers.

Milestone — A zero-duration node in the activity network marking a significant integration or delivery point. Milestones anchor earned-value tracking and provide natural reporting boundaries.

Project Design — An architecture-driven methodology for project planning that derives schedule, cost, and risk estimates from the structural decomposition of the system to be built, rather than from bottom-up task guessing.

Resource Leveling — Adjusting the schedule to avoid resource over-allocation by shifting non-critical activities within their available float. Resource leveling preserves the critical-path duration while smoothing peak demand.

Risk Index — A measure of schedule risk based on the distribution of float across the activity network and the density of near-critical paths. A project with many low-float paths has a higher risk index than one with a single clear critical path.

S-Curve — The characteristic shape of cumulative earned value over time in a well-planned project.

Sub-Critical Staffing — Resource levels insufficient to staff the critical path, forcing serial execution and dramatically increasing risk.

Time-Cost Curve — A plot of all feasible project solutions showing the relationship between schedule duration and total cost.

Work Package — A unit of work derived from an architectural component, sized to be estimable by a single developer or small team. Work packages are the nodes of the activity network.

Appendix B: Applicability Checklist

Project design is applicable to any project, but is particularly valuable for projects that:

Have an identified architectural decomposition or can invest in creating one

Require accurate cost and schedule estimates for stakeholder commitment

Involve multiple developers or teams working on interconnected components

Face schedule pressure that demands understanding of compression limits

Have experienced estimation failures on prior similar projects

Must present options to management for resource commitment decisions

Operate under regulatory, compliance, or contractual obligations that require defensible planning

Appendix C: Key Formulas

Forward Pass: ES(i) = max(EF of all predecessors); EF(i) = ES(i) + Duration(i)

Backward Pass: LF(i) = min(LS of all successors); LS(i) = LF(i) - Duration(i)

Total Float: TF(i) = LS(i) - ES(i)

Free Float: FF(i) = min(ES of all successors) - EF(i)

Criticality Risk Index: Risk = (4C + 3R + 2Y + 1G) / (4N)

Activity Risk Index: Risk = 1 - sum(Fi + 1) / (N * M)

Compression Factor: Duration = 0.7 * Normal; Cost = 1.8 * Normal

Project Cost: Cost = integral of staffing curve over time

Efficiency: Efficiency = sum of activity durations / (staffing * total duration)

References and Influences

The concepts presented in this paper are grounded in established work in project management, critical path analysis, and software engineering. Project design is not presented as a novel invention, but as a practitioner-oriented articulation of principles that have emerged through decades of practice.

Juval Löwy

Löwy, Juval. Righting Software. Addison-Wesley, 2019.

Löwy's work is the primary foundation for this articulation of project design. His IDesign methodology integrates system architecture with project planning, treating design as the prerequisite for estimation. The activity derivation from architecture, the Feed Me or Kill Me decision framework, the time-cost curve, the risk quantification models, the staffing principles, and the earned value validation approach described in this paper are derived from IDesign training. This paper consolidates these principles into a single cohesive reference.

Critical Path Method (CPM)

Kelley, James E.; Walker, Morgan R. "Critical-Path Planning and Scheduling." Proceedings of the Eastern Joint Computer Conference, 1959.

The Critical Path Method, developed independently by DuPont and the U.S. Navy in the late 1950s, provides the mathematical foundation for network analysis used throughout this paper. The forward and backward pass algorithms, float calculations, and the concept of the critical path as the project duration constraint are direct applications of CPM.

Earned Value Management

Fleming, Quentin W.; Koppelman, Joel M. Earned Value Project Management. Fourth Edition. Project Management Institute, 2010.

Earned value management provides the tracking and projection techniques used during project execution. The S-curve validation during planning and the three-line tracking during execution are applications of EVM adapted for software project contexts.

Bent Flyvbjerg

Flyvbjerg, Bent. "Over Budget, Over Time, Over and Over Again: Managing Major Projects." Oxford Handbook of Project Management, 2011.

Flyvbjerg's research on megaproject performance provides the empirical basis for the discussion of large project fragility and the network of networks approach.

Frederick P. Brooks Jr.

Brooks, Frederick P. The Mythical Man-Month. Addison-Wesley, 1975.

Brooks's observation that adding people to a late project makes it later is reflected throughout this paper's treatment of staffing, compression limits, and the relationship between team size and communication overhead.

Nassim Nicholas Taleb

Taleb, Nassim Nicholas. Antifragile: Things That Gain from Disorder. Random House, 2012.

Taleb's framework for antifragility informs the discussion of project design as an asymmetric investment — small capped cost with potentially large payoffs — and the principle that projects should be designed to benefit from variability rather than merely resist it.

Author's Note

Project Design, including its integration of architectural decomposition with project planning, the multi-option decision framework, and the risk quantification techniques, originates from Juval Löwy's IDesign methodology. This paper does not

introduce a new project management approach. It provides a consolidated, practitioner-oriented articulation that integrates established project management techniques with architecture-first planning, emphasizing the complete activity inventory — including the many non-code activities that are routinely underestimated — and the structured decision-making process.

The intent of this paper is to serve as a durable reference that translates project design principles into a form suitable for consistent application, discussion, and review within modern engineering organizations.

Distribution Note

This document is provided for informational and educational purposes. It may be shared internally within organizations, used as a reference in project planning discussions, or adapted for non-commercial educational use with appropriate attribution. This paper does not represent official policy, standards, or project management mandates of any current or former employer. All examples are generalized and abstracted to avoid disclosure of proprietary or sensitive information.

APPENDIX F

Compiled Context Runtime

Compiled Context Runtime

Process-Driven Agent Execution with Unbounded Local Memory

Author: William Christopher Anderson **Date:** April 2026 **Version:** 1.0

✦ ✦ ✦

Executive Summary

Large language models are stateless. Every call begins from nothing. The entire burden of continuity — what happened before, what matters now, what the system has learned — falls on whatever context is stuffed into the prompt window. Today's agent systems respond to this constraint with brute force: they pack as much

raw text as possible into every call, hope the model attends to the right parts, and accept that the model forgets everything between sessions.

This approach is simultaneously expensive and unreliable. It is expensive because every token sent to the model incurs cost, and most of those tokens are irrelevant to the current task. It is unreliable because the model has no mechanism to distinguish signal from noise in a bloated context window — the important instruction on line 400 competes for attention with the boilerplate on line 12.

The Compiled Context Runtime (CCR) is an architectural model that eliminates both problems. It introduces three structural innovations:

1. **Process definitions** — Agent workflows codified as versioned, executable YAML specifications. Each process declares its steps, gates, knowledge requirements, and trigger conditions. The agent's creativity goes into executing the steps, not remembering them.
2. **Compiled context injection** — A compilation pipeline that retrieves relevant knowledge, compresses it into a lossless format (CTX), and injects only what is needed for the current process step. The context window receives precision-compiled packages, not raw text dumps.

3. **Memory and context chains** — Persistent, linked data structures in a local database that capture the full history of agent interactions, decisions, corrections, and execution outcomes. Chains compile into CTX packages on demand, giving the model access to effectively unlimited historical depth while staying within the token window.

The consequence is a system where the context window is no longer a hard limit. It becomes a viewport — a precision-scoped lens into a local store of potentially millions of memories, thousands of execution records, and hundreds of thousands of embeddings. The model sees exactly what it needs for the current step. Nothing more. Nothing less.

The economic implications are significant. By reducing input tokens per task by approximately 88% and eliminating exploratory calls through deterministic process execution, the CCR model cuts LLM API costs by an order of magnitude. At enterprise scale, this represents millions of dollars in annual savings per organization. At global scale — across the hundreds of millions of knowledge workers, analysts, researchers, writers, and developers adopting LLM-assisted workflows — the aggregate savings exceed billions of dollars annually.

This paper describes the architectural model, the compilation pipeline, the memory system, the learning loop that makes processes and context progressively more efficient, and the economic analysis that quantifies the impact.

✦ ✦ ✦

Abstract

Current approaches to LLM-based agent systems treat the context window as a fixed-size container into which raw text is packed before each inference call. This produces three systemic failures: excessive token cost from irrelevant context, unreliable model behavior from attention dilution, and complete memory loss between sessions. The Compiled Context Runtime addresses these failures through process-driven execution (codified workflows that eliminate prompt-dependent behavior), compiled context injection (a pipeline that retrieves, compresses, and scopes knowledge to the current step), and persistent memory chains (linked data structures that give the model access to unbounded historical depth through precision compilation). This paper presents the architectural model, the compilation format, the memory and context chain data structures, the process discovery and refinement loop, and a quantitative analysis of token economics at individual, enterprise,

and global scale. The system is local-first by design: all data — process definitions, execution history, knowledge embeddings, compiled context packages — resides on the user's machine. No workflow data crosses a network boundary except the compiled context injected into the LLM inference call itself.

✦ ✦ ✦

1. Introduction

1.1 The Statelesness Problem

Large language models are functions. They accept a sequence of tokens and produce a sequence of tokens. They retain nothing between calls. Every inference begins from a blank state, and whatever continuity the system exhibits must be constructed entirely from the input context.

This is a fundamental architectural constraint, and the industry's response to it has been remarkably uniform: pack more into the context window. Conversation history is appended. Retrieval-augmented generation (RAG) inserts document fragments. System prompts grow to thousands of tokens of instructions. The result is a context window that serves simultaneously as instruction manual,

conversation log, knowledge base, and working memory — a single undifferentiated buffer asked to do the work of four distinct systems.

The consequences are predictable. Important instructions are buried among retrieved passages. Relevant history competes with irrelevant history for the model's attention. Token costs scale linearly with the amount of context stuffed into each call, regardless of how much of that context is actually used. And when the session ends, everything is lost.

1.2 The Agent Amplification

Agent systems amplify every failure mode. An agent is not a single inference call — it is a sequence of calls, each building on the last, often spanning hours of work. An agent reviewing a pull request might make twenty calls: reading files, understanding context, analyzing changes, composing feedback. At each call, the agent system must reconstruct the relevant context from scratch, because the model remembers nothing from the previous call.

The common solution is to carry forward the entire conversation history. This means that call twenty contains the full transcript of calls one through nineteen — most of which is irrelevant to the current task of composing a final review comment. The token cost of the twentieth call dwarfs its informational content.

More critically, the agent has no structured memory. It cannot recall what it learned three sessions ago. It cannot look up a decision it made last week. It cannot walk a chain of related corrections to understand the current state of a preference. Every session begins from whatever fits in the system prompt, and everything else is gone.

1.3 The Compiled Context Alternative

The Compiled Context Runtime (CCR) inverts the relationship between the model and its context. Instead of the context window being a container that the system fills, it becomes a viewport that the runtime controls.

The runtime maintains three independent systems:

- A **process engine** that defines agent workflows as executable specifications, eliminating the need for the model to remember what to do
- A **compilation pipeline** that transforms raw knowledge into compressed, scoped packages, eliminating the need to stuff raw text into the context
- A **memory system** that persists, links, and indexes every interaction across sessions, eliminating the assumption that the model must forget

These three systems compose to produce a model of agent execution where the context window is used surgically — receiving only what the current step requires — while the actual depth of available context is limited only by local storage.

1.4 Model-Agnostic by Construction

The CCR is not coupled to any specific language model. Compiled CTX packages are plain text — any model that accepts text input can consume them. Process definitions are YAML — they describe what to do, not how any particular model should do it. Memory chains are data structures — they store and retrieve knowledge independently of which model uses it.

Critically, the model is not statically configured — it is **dynamically selected**. When a step in a process needs execution, the runtime evaluates the task requirements (reasoning depth, code generation, speed constraints, data sensitivity), checks available models and their capabilities, and selects the optimal model for that specific step. The process definition does not say “use Claude” or “use GPT” — it describes the work, and the runtime matches the work to the best available model. This means:

- **Dynamic model selection** — The agent evaluates each task, checks what models are available and what they’re good at, and picks the right one. A

complex architectural decision routes to the most capable reasoning model. A simple file transformation routes to a fast, cheap model. A step handling sensitive data routes to a local model that never leaves the machine. This happens automatically, per-step, without human intervention.

- **Cross-model intelligence** — Because knowledge lives in compiled context packages and memory chains — not in any model's weights — intelligence accumulates across model boundaries. A decision made by Claude gets recorded in a memory chain. That memory chain gets compiled into context for a step executed by GPT. The insight transfers. The intelligence is in the data layer, and every model that touches it gets smarter.
- **Survive model obsolescence** — When a better model launches, the CCR's accumulated knowledge, processes, and execution history carry forward unchanged. Nothing is lost to a model transition. The new model immediately benefits from everything every previous model learned, because it's all in the compiled context.
- **No vendor lock-in** — The value accrues in the local data layer (processes, memories, knowledge), not in the model. The model is a replaceable inference endpoint. The intelligence is

in the compiled context. Switch providers, switch models, switch architectures — the accumulated intelligence persists.

1.5 Local-First as Architectural Requirement

The CCR model is local-first by design, not by preference. This is an architectural requirement, not a deployment choice.

Process definitions encode an organization's workflows. Execution history records what an agent has done and learned. Memory chains capture every decision, correction, and preference accumulated over months of use. Knowledge embeddings index proprietary content, internal documentation, and domain-specific reference material.

None of this data should cross a network boundary. It is operationally sensitive, competitively valuable, and privacy-critical. The only data that leaves the user's machine is the compiled context package injected into the LLM inference call — and that package contains only what the current step requires, compiled into a format that strips structural metadata.

Local-first is what makes the system trustworthy. If the memory system required shipping data to a cloud service, adoption would be structurally limited to organizations willing to externalize their workflows. Local-first removes that constraint entirely.

2. The Five Primitives

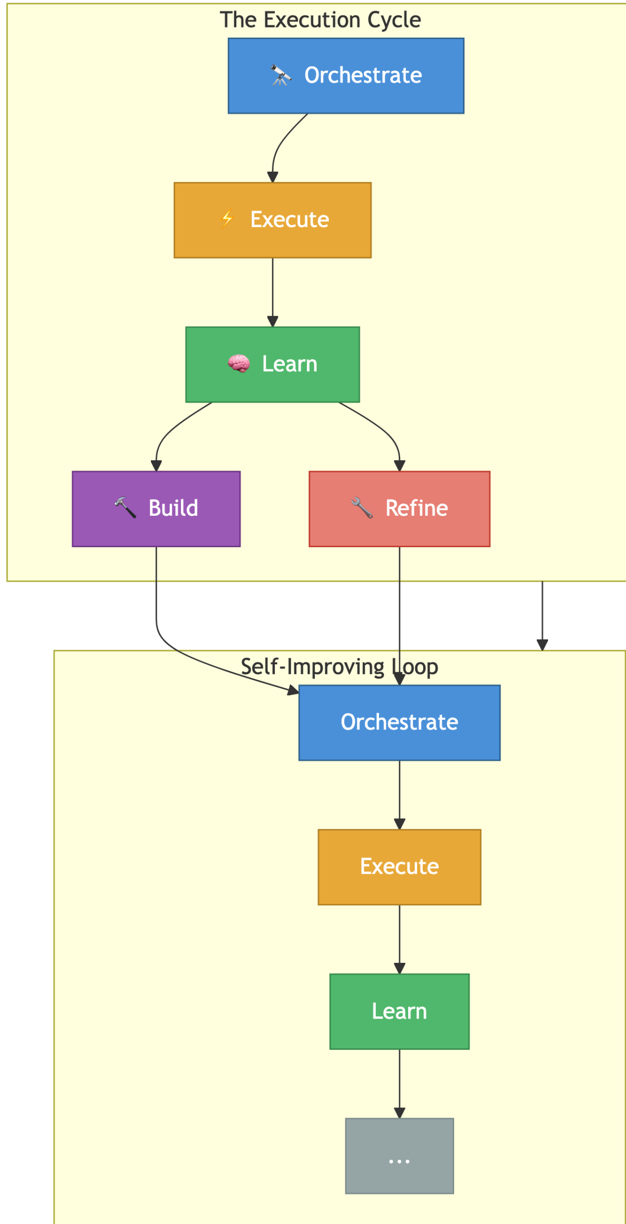
2.1 The Execution Cycle

Before defining how processes are represented, the CCR establishes the fundamental cycle that governs all agent work. Every action an agent takes is an instance of one of five primitives, executed in cycle:

1. **Orchestrate** — Invoke meta-learning. Pull the latest state. Read the knowledge index. Look up relevant knowledge by topic. Compile context. Analyze dependencies. Decompose the task. Dispatch.
2. **Execute** — Do the work. Write code, configure systems, run tests, produce artifacts. This is the only primitive that produces external output.
3. **Learn** — Analyze outcomes at two levels:
    - **Meta-learning:** Evaluate the processes themselves — execution patterns, recovery strategies, failure modes. Update directives and process definitions.
    - **Context-learning:** Evaluate the domain — what was discovered about the subject matter, the working environment, the user's preferences. Update knowledge and memory chains.

4. **Build** — Create new processes, knowledge artifacts, or tools when Learn identifies gaps. A repeated ad-hoc sequence becomes a process definition. A missing knowledge topic becomes a new entry. A missing capability becomes a new tool.
5. **Refine** — Improve existing processes, knowledge, and tools when Learn identifies weaknesses. A slow step gets optimized. A stale knowledge reference gets updated. A process gate that fails too often gets its preconditions adjusted.

The cycle: Orchestrate → Execute → Learn → Build/Refine (if needed) → Orchestrate (better)


The Execution Cycle
🔭 Orchestrate
⚡ Execute
🧠 Learn
🔨 Build
🔧 Refine
Self-Improving Loop
Orchestrate
Execute
Learn
...

diagram

2.2 Why Five Primitives

The five primitives are not arbitrary. They are the minimal set required for a self-improving execution system:

- Without **Orchestrate**, the agent has no context and works blind.
- Without **Execute**, no work is produced.
- Without **Learn**, the agent repeats mistakes and never improves.
- Without **Build**, gaps in processes and knowledge persist indefinitely.
- Without **Refine**, existing processes degrade as conditions change.

Remove any one and the system loses a critical capability. Add a sixth and it can be expressed as a composition of the existing five. The primitives are orthogonal and complete.

2.3 Processes Formalize the Cycle

Every process definition in the CCR is a codification of the five primitives applied to a specific workflow:

- The process's **knowledge references** and **gates** are the Orchestrate phase — ensuring context is loaded and preconditions are met before work begins.
- The process's **steps** are the Execute phase — the actual work, performed in sequence.
- The process's **execution recording** is the Learn phase — capturing what happened for later analysis.
- The **process discovery** system is the Build phase — detecting new patterns and proposing new process definitions.
- The **process refinement** system is the Refine phase — analyzing execution records and proposing improvements.

The five primitives are the theory. Process definitions are the implementation. The CCR makes the cycle explicit, executable, and self-improving.

✦ ✦ ✦

3. Process Definitions

3.1 Processes as Data, Not Prompts

The first structural innovation of the CCR is the separation of workflow definition from workflow execution.

In conventional agent systems, the workflow lives in the prompt. A system prompt might instruct the agent: "First, check CI status. Then read the failing test. Then fix the test. Then run the test suite. Then commit." The agent follows these instructions — if it attends to them, if they fit in the context window, if it doesn't hallucinate an alternative sequence.

In the CCR, the workflow is a data structure:

```yaml
process: fix_ci_failure
version: 3
trigger:
  type: event
  match:
    source: ci
    status: failure

knowledge:
  - engineering.testing
  - project.ci_pipeline

gates:
  - execution_context_exists
  - branch_clean

steps:
  - id: read_failure
    action: read_ci_log
    description: Identify the failing test and
        error message

  - id: locate_source
    action: find_relevant_code

        description: Find the source code
        responsible for the failure

  - id: diagnose
    action: analyze_failure
    description: Determine root cause of the
        failure

  - id: implement_fix
    action: write_code
    description: Implement the fix

  - id: verify
```

```
    action: run_tests

        description: Run the test suite to verify
        the fix

  - id: commit
    action: commit_and_push
    description: Commit the fix and push
    gates:
      - tests_pass
```

This definition is stored in a database, versioned, and executable. The runtime reads it and executes each step in sequence. The model is invoked at each step with exactly the context that step requires — not a prompt full of instructions it might or might not follow.

3.2 Gates

Gates are preconditions evaluated before execution begins or before individual steps execute. They are binary — pass or fail — and their failure halts the process with a recorded reason.

Gates serve two purposes. First, they prevent the agent from executing in invalid states — attempting to commit when tests are failing, or beginning work without an execution context. Second, they create a verifiable execution contract. A process with three gates and six steps produces a deterministic sequence of checkpoints that can be audited after the fact.

3.3 Knowledge References

Each process declares which knowledge topics it needs. The runtime resolves these references against the knowledge store before execution begins. This is not retrieval-augmented generation — it is declarative context scoping. The process author specifies exactly what the model should know for this workflow. The runtime compiles it. The model receives it.

This eliminates the two failure modes of RAG: retrieving irrelevant passages (because the process author specified exactly what's needed) and missing relevant passages (because the knowledge references are explicit and verified at process definition time).

3.4 Process Inheritance and Composition

Process definitions are object-oriented. A process can **extend** another process, inheriting its steps, gates, and knowledge references while overriding or adding to them. This is structural inheritance — the same concept as class inheritance in Java or C#, applied to workflow definitions.

```yaml
process: fix_ci_failure_with_notification
version: 1
extends: fix_ci_failure

# Inherits all steps, gates, knowledge from
        fix_ci_failure
# Adds a notification step after commit
steps:
  - inherit: all
  - id: notify
    action: send_notification
    description: Notify the team that the CI
        failure has been fixed
    after: commit

# Adds additional knowledge ref
knowledge:
  - inherit: all
  - team.notification_preferences
```

The inheritance model supports:

- **Single inheritance** — A process extends exactly one parent. The parent's steps, gates, and knowledge references are inherited unless explicitly overridden.
- **Step override** — A child process can replace a parent step by declaring a step with the same ID. The parent's version is discarded; the child's version is used.

- **Step insertion** — A child can insert steps before or after inherited steps using `before:` and `after:` directives. The parent's sequence is preserved; the child's additions are spliced in.
- **Gate extension** — A child inherits all parent gates and can add additional gates. Gates cannot be removed — a child process is always at least as constrained as its parent.
- **Knowledge extension** — Knowledge references compose. A child inherits all parent knowledge and can add more. This ensures the child always has at least as much context as the parent.
- **Abstract processes** — A process can be declared `abstract: true`, meaning it cannot be executed directly but serves as a template for concrete processes. This is the process equivalent of an abstract class.

```yaml
# Abstract base process — cannot execute directly
process: standard_code_change
abstract: true
version: 1

gates:
  - execution_context_exists
  - branch_clean

knowledge:
  - engineering.pull_request
  - project.code_conventions

steps:
  - id: analyze
    action: analyze_requirements
    abstract: true     # Must be overridden by
        child

  - id: implement
    action: write_code
    abstract: true     # Must be overridden by
        child

  - id: verify
    action: run_tests

  - id: commit
    action: commit_and_push
    gates:
      - tests_pass
```

Concrete processes extend this base:

```
process: fix_bug
extends: standard_code_change
version: 1

steps:
  - id: analyze
    action: read_bug_report
    description: Identify root cause from bug
        report and logs

  - id: implement
    action: write_fix
    description: Implement the minimal fix

---

process: add_feature
extends: standard_code_change
version: 1

knowledge:
  - inherit: all
  - engineering.design_review

steps:
  - id: analyze
    action: read_feature_spec
    description: Understand the feature
        requirements

  - id: implement
    action: write_feature
    description: Implement the feature with tests
```

This is polymorphism applied to workflows. A `standard_code_change` defines the contract — what gates must pass, what knowledge is loaded, what

sequence is followed. Concrete processes fill in the domain-specific behavior. The runtime doesn't care whether it's executing `fix_bug` or `add_feature` — it executes the linked process, step by step, through the same pipeline.

3.5 Process Interfaces

Just as object-oriented systems separate interface from implementation, the CCR separates process **contracts** from process **implementations**. A process interface defines what a process must do — its required steps, gates, and knowledge references — without specifying how.

```
interface: code_change
version: 1
description: Contract for any process that
        modifies code

required_gates:
  - execution_context_exists
  - branch_clean

required_steps:
  - id: analyze
    description: Understand what needs to change
  - id: implement
    description: Make the change
  - id: verify
    description: Verify the change works

required_knowledge:
  - engineering.pull_request
```

Any process that declares `implements: code_change` must provide concrete definitions for all required steps. The compiler verifies this at compile time — a process that claims to implement an interface but is missing a required step fails to compile.

```yaml
process: fix_bug
version: 1
implements: code_change

# Compiler verifies: analyze, implement, verify
        steps all present
# Compiler verifies: execution_context_exists,
        branch_clean gates present
# Compiler verifies: engineering.pull_request in
        knowledge refs

steps:
  - id: analyze
    action: read_bug_report
    description: Identify root cause from bug
        report and logs

  - id: implement
    action: write_fix
    description: Implement the minimal fix

  - id: verify
    action: run_tests
    description: Run the test suite
```

Process interfaces enable:

- **Substitutability** — Any process implementing the `code_change` interface can be used where a `code_change` is expected. The runtime can

dynamically select which concrete process to execute based on the trigger event, the project context, or user preference.

- **Contract verification** — The compiler guarantees that every implementing process satisfies the interface contract. Missing steps, missing gates, missing knowledge references are compile-time errors.
- **Organizational standards** — An organization defines process interfaces that encode their standards: "every code change must include analysis, implementation, and verification." Teams provide concrete implementations that fit their specific workflows. The interface ensures consistency; the implementation allows flexibility.
- **Composability** — A process can implement multiple interfaces, satisfying multiple contracts simultaneously. A `deploy_hotfix` process might implement both `code_change` and `deployment`, ensuring it meets the standards for both workflows.

This is the Interface Segregation Principle applied to processes. Interfaces are small, focused contracts. Processes implement the ones relevant to their domain. The compiler enforces the contracts. The runtime dispatches polymorphically.

3.6 The Process Compiler

Process definitions are not interpreted — they are **compiled**. The compilation pipeline is analogous to class loading in the JVM or assembly loading in the CLR: YAML source is parsed, validated, linked, and emitted as an executable runtime object.

Compilation stages:

1. **Parse** — YAML source is deserialized into a raw ProcessDefinition AST (abstract syntax tree). Syntax errors are caught here — malformed YAML, missing required fields, invalid types.
2. **Validate** — The AST is validated against the process schema. Semantic errors are caught: duplicate step IDs, circular inheritance, references to nonexistent gates, abstract steps that aren't overridden, knowledge references that don't resolve. Validation produces a list of errors and warnings. A process with errors cannot proceed to linking. Warnings are recorded but do not block compilation.
3. **Resolve inheritance** — If the process extends a parent, the compiler loads the parent (recursively, for chains of inheritance), merges inherited steps/gates/knowledge with the child's overrides, and verifies that all abstract steps have been implemented.

4. **Link** — Symbolic references are resolved to concrete objects. Knowledge topic names are resolved to file paths. Gate names are bound to evaluator functions. Step actions are bound to handler callables. The result is a `LinkedProcess` — an object where every reference is a direct pointer, not a name to be looked up at runtime. This is the process equivalent of a linked executable.
5. **Emit** — The LinkedProcess is registered in the process table and cached. It is ready for execution. The compiled form is stored alongside the source YAML, so recompilation is only needed when the source changes.

Compile-time guarantees:

Because processes are validated at compile time, the runtime can make guarantees that interpreted systems cannot:

- Every knowledge reference resolves to a real file
- Every gate references a registered evaluator
- Every step action references a registered handler
- Inheritance chains are acyclic
- Abstract steps are fully implemented
- No duplicate step IDs exist
- Required fields are present and correctly typed

A process that compiles will not fail due to structural errors at runtime. Runtime failures are limited to actual execution issues — a test that fails, a file that's missing, an API that's down. The structural integrity is guaranteed by the compiler.

3.7 Versioning and Evolution

Every modification to a process creates a new version. Execution records link to the version that was active at execution time. This produces a complete audit trail: which version of which process produced which outcome, with which knowledge references, at which time.

Version history enables the refinement loop described in Section 8.

✦ ✦ ✦

4. The Runtime

4.1 A Managed Runtime for Agent Processes

The Compiled Context Runtime is a **managed runtime** in the same sense as the JVM or the CLR. It is not a script runner — it is a full execution environment that manages the lifecycle of process objects, provides memory management with garbage collection,

implements multi-level caching, offers observability through tracing and debugging, and is extensible through a messaging bus.

The analogy is precise:

JVM/CLR Concept	CCR Equivalent
Class	ProcessDefinition (YAML source)
Class loader	ProcessLoaderEngine (YAML parse + validate)
Linker	ProcessLinkerEngine (resolve refs, bind gates)
Loaded class	LinkedProcess (all refs resolved)
Object instance	ExecutionRecord (a running/ completed execution)
Garbage collector	GCManager (generational, mark-sweep)
JIT cache	CacheManager (L1/L2/L3 tiered)
Class hierarchy	Process inheritance (extends, abstract)
Interface	Gate contracts + step action contracts
Bytecode verifier	Process validator (compile-time guarantees)
Debugger	Execution tracer + step inspector
ClassNotFoundException	ProcessLoadError
LinkageError	LinkError (unresolved ref)

4.2 The Caching System

The CCR implements a three-tier cache modeled on CPU cache hierarchies:

L1 — In-Memory Hot Cache. Recently compiled CTX packages, recently linked processes, and recently resolved knowledge topics. Access time: microseconds. Size: bounded by memory (configurable, default 256MB). Eviction policy: adaptive replacement cache (ARC) — balances recency and frequency. This is where the runtime looks first for any compiled artifact.

L2 — SQLite Warm Cache. Compiled artifacts that have been evicted from L1 but are still likely to be needed. Serialized to disk in a SQLite database. Access time: single-digit milliseconds. Size: bounded by disk (configurable, default 2GB). Eviction policy: time-aware LFU — items that haven't been accessed within a configurable window are evicted. Promotion to L1 occurs on access.

L3 — Cold Storage. Full compilation artifacts archived for historical reference. This tier is not accessed during normal execution — it exists for auditing and recompilation. Items promoted from L3 go to L2 first, then L1 on access.

Cache warming. On startup, the runtime warms the cache by preloading the most frequently used processes and their knowledge references. The warming strategy is derived from execution history —

processes executed most often in the last 30 days are preloaded. This means the first execution after startup is nearly as fast as subsequent ones.

4.3 Generational Garbage Collection

The CCR manages a large volume of runtime objects: memory nodes, context chains, execution records, compiled CTX packages, cached compilation artifacts. Not all of these need to persist forever. The generational garbage collector reclaims objects that are no longer reachable, following the same generational hypothesis as the JVM: most objects die young.

Three generations:

- **Gen 0 (Nursery)** — Newly created objects: fresh memory nodes, in-progress execution records, temporary CTX compilations. Collected frequently (every N allocations or every M minutes). Most objects die here — a temporary compilation for a single step is used once and discarded.
- **Gen 1 (Survivor)** — Objects that survived one or more Gen 0 collections. These have demonstrated some persistence — a memory node that's been referenced by another node, an execution record that's been finalized, a CTX package that's been accessed multiple times. Collected less frequently.

- **Gen 2 (Tenured)** — Long-lived objects: established memory chains, frequently-accessed knowledge packages, historical execution records marked for retention. Collected rarely. Objects in Gen 2 are the permanent knowledge base — the accumulated expertise described in Section 6.

Collection algorithm: Mark-sweep with reference counting. The collector identifies root objects (active execution contexts, pinned memory chains, cached processes), traces all reachable objects from roots, and sweeps unreachable objects. Reference counts provide fast detection of isolated garbage; the full mark-sweep handles cycles.

Promotion criteria: An object is promoted from Gen N to Gen N+1 when it survives a configurable number of collections (default: 2 for Gen 0→1, 5 for Gen 1→2). Objects can also be explicitly promoted (pinned) by the user or by the runtime when they're referenced by a long-lived chain.

4.4 Observability

A runtime without observability is a black box. The CCR provides full instrumentation for debugging, tracing, and monitoring:

Execution tracing. Every process execution produces a trace — a structured record of every step executed, every gate evaluated, every knowledge reference resolved, every CTX package compiled,

every model invocation made, and every outcome recorded. Traces are linked to execution contexts and stored in the execution record. They can be inspected after the fact to understand exactly what happened and why.

Step-level debugging. The runtime supports breakpoints at the step level. A step can be marked as a breakpoint in the process definition or at runtime. When a breakpoint step is reached, execution pauses, and the current state is surfaced: the compiled context that would be injected, the gate results, the execution history so far. The user can inspect, modify context, or resume.

Structured logging. All runtime events are emitted as structured log entries with correlation IDs that link to the active execution context. Log levels: TRACE (every internal operation), DEBUG (compilation and linking details), INFO (step execution, gate results), WARN (non-fatal issues), ERROR (step failures, gate failures).

Metrics. The runtime exposes metrics for monitoring: - Cache hit rates per tier (L1/L2/L3) - GC pause times and collection counts per generation - Compilation times (parse, validate, link, emit) - Token usage per step and per process - Execution duration per step - Model selection decisions and latency - Memory pressure and allocation rates

Diagnostic commands. The CLI exposes diagnostic tools: - `neuron trace <execution-id>` — full execution trace - `neuron cache stats` — cache hit rates, sizes, eviction counts - `neuron gc stats` — generation sizes, collection history, promotion rates - `neuron process inspect <name>` — compiled process details, inheritance chain - `neuron memory inspect <chain-id>` — memory chain visualization

4.5 Bus Extensibility

The runtime is extensible because it is built on a **messaging bus**. Every component in the system communicates through typed messages on the bus. The runtime itself does not call components directly — it publishes events, and components subscribe to the events they care about.

This means the runtime is open for extension without modification:

- **Custom step handlers** — Register a new action type by subscribing to `step.execute` events where `action` matches your handler. The runtime doesn't need to know about your handler — it publishes the event, your handler responds.
- **Custom gate evaluators** — Register a new gate by subscribing to `gate.evaluate` events where `gate_name` matches your evaluator. Same pattern.

- **Custom model providers** — Register a new LLM provider by subscribing to `model.invoke` events. The model selection engine routes to your provider based on selection criteria.
- **Custom observability** — Subscribe to `trace.*` events to build custom dashboards, export to external systems, or integrate with existing APM tools.
- **Plugins** — The plugin system is built on the bus. A plugin is a bundle of event subscriptions with a manifest. Loading a plugin registers its subscriptions. Unloading a plugin removes them. No code changes to the runtime.

The bus scales from in-process (single agent) to IPC (multi-agent on one machine) to network (distributed agents). The same subscription model works at every scale because the message format is uniform and the delivery mechanism is pluggable.

4.6 The Process IDE

Because processes are compiled with full validation, the compilation pipeline can power developer tooling:

Real-time validation. As a user edits a process YAML file, the compiler runs continuously, surfacing errors and warnings inline — missing knowledge references, unresolved gates, inheritance conflicts,

abstract steps that need implementation. This is the process equivalent of a TypeScript language server providing red squiggles as you type.

Autocomplete. The compiler knows the full schema, all registered gates, all registered actions, all knowledge topics in the index. It can provide autocomplete suggestions for every field in a process definition.

Inheritance visualization. For processes that extend other processes, the IDE can show the resolved inheritance chain — which steps are inherited, which are overridden, which knowledge references come from which ancestor. This is the process equivalent of a class hierarchy viewer.

Execution dry-run. The IDE can simulate process execution without invoking the LLM — evaluating gates against current state, resolving knowledge references, computing the viewport allocation, and showing exactly what context would be injected at each step. This lets process authors validate their workflows before committing them.

Diff and history. Process versions are stored with full history. The IDE can show diffs between versions, highlight what changed, and correlate version changes with execution outcome changes from the refinement engine.

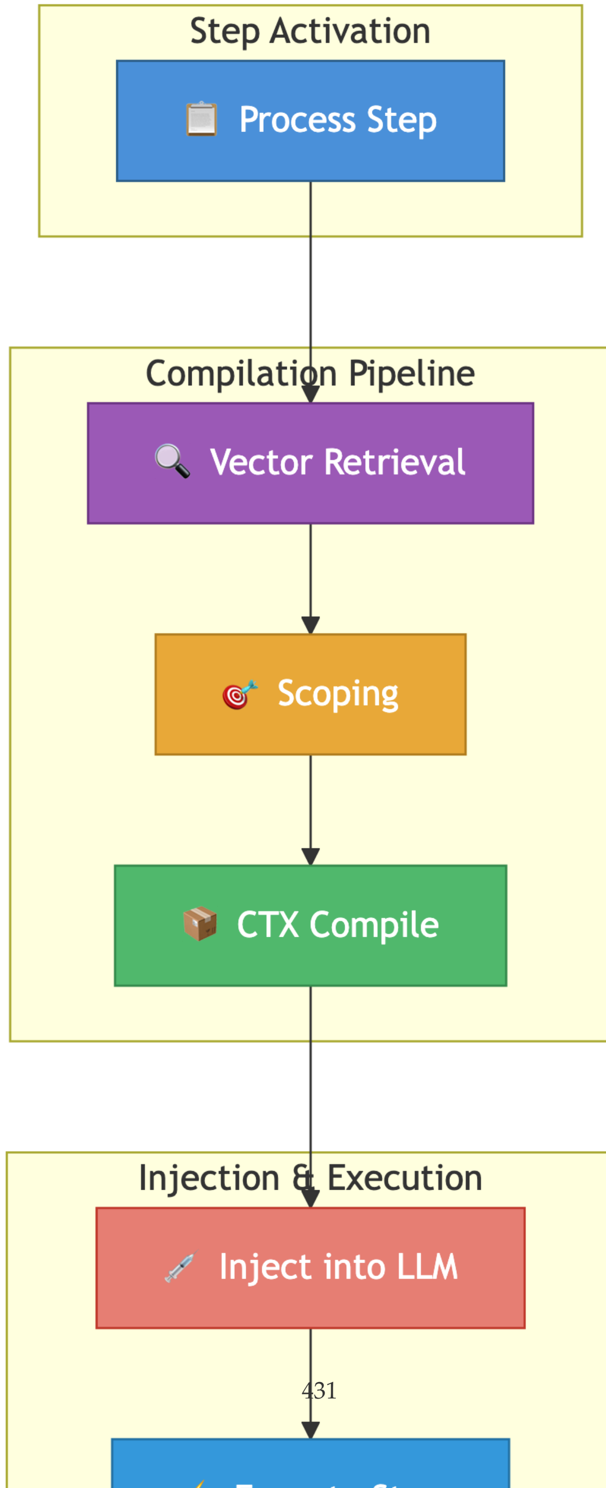
The Process IDE is not a separate product — it is a natural consequence of the compiler architecture. Any system that compiles with full validation can power tooling. The CCR's compiler produces the same kind of structured output (AST, error list, resolved symbols) that a language compiler produces, and the same kinds of tools can be built on top of it.

✦ ✦ ✦

5. Compiled Context Injection

5.1 The Compilation Pipeline

The CCR compilation pipeline transforms raw knowledge and historical context into compressed, scoped packages injected into the model at each process step.


Step Activation
Process Step
Compilation Pipeline
Vector Retrieval
Scoping
CTX Compile
Injection & Execution
Inject into LLM

diagram

The pipeline operates in four stages:

1. **Retrieval** — The process step's knowledge references are resolved against the local knowledge store. Memory chains and context chains relevant to the current task are retrieved via vector similarity search.
2. **Scoping** — Retrieved content is filtered to what the current step actually needs. A six-step process does not carry step one's context through step six unless the process definition explicitly requires it.
3. **Compilation** — Scoped content is compiled into CTX format — a lossless semantic compression that preserves all meaning while reducing token count. The compilation is structural: redundant framing is removed, cross-references are resolved inline, and hierarchical relationships are encoded in a compact notation.
4. **Injection** — The compiled CTX package is placed into the model's context window alongside the step-specific instructions. The model receives a single, coherent, compressed context that contains exactly what it needs.

5.2 The CTX Format

The CTX format is a lossless compression scheme for structured knowledge. It was developed independently for compiling research whitepapers into compact reference formats and has been validated across documents ranging from 5,000 to 30,000 words.

The format achieves 40-60% token reduction on narrative text and 60-84% reduction on structured knowledge (tables, hierarchies, reference material). The compression is lossless in the sense that all semantic content is preserved — a model consuming the CTX version of a document has access to the same information as a model consuming the original, but at a fraction of the token cost.

The format is not a general-purpose compression algorithm. It is specifically designed for LLM consumption: the output is valid text that the model can read directly. No decompression step is required. The model simply reads a more compact representation of the same information.

5.3 Per-Step Scoping

The most significant cost reduction comes not from compression but from scoping. A conventional agent system might inject 50,000 tokens of context into every call — the full conversation history, the full retrieved documents, the full system prompt. The CCR injects only what the current step needs.

Consider a six-step process where each step requires different knowledge:

Step	Knowledge Needed	Compiled Size
Read CI log	CI pipeline docs	1,200 tokens
Locate source	Project structure	2,400 tokens
Diagnose	Testing standards	1,800 tokens
Implement fix	Code conventions	3,200 tokens
Run tests	Test commands	800 tokens
Commit	Git workflow	600 tokens

Average context per step: 1,667 tokens. Total across six steps: 10,000 tokens. A conventional system would inject the same 50,000-token context six times: 300,000 tokens. The CCR uses 97% fewer input tokens for the same workflow.

✦ ✦ ✦

6. Memory and Context Chains

6.1 The Memory Problem

The context window is ephemeral. When a session ends, the model's state is destroyed. Any knowledge accumulated during the session — corrections, preferences, decisions, learned context — is lost unless explicitly persisted somewhere external.

Current approaches to persistence are primitive. Some systems append to a markdown file. Others maintain a flat key-value store. None preserve the structure of how memories relate to each other: which correction superseded which earlier belief, which decision led to which outcome, which preference was refined through which sequence of interactions.

6.2 Memory Chains

A memory chain is a linked sequence of related memory nodes stored in a relational database. Each node contains:

- **Content** — The memory itself (a decision, preference, correction, observation)
- **Type** — Classification (correction, decision, preference, observation, outcome)
- **Links** — Typed edges to other nodes (supersedes, refines, contradicts, led_to, caused_by)
- **Embedding** — Vector representation for similarity search
- **Metadata** — Timestamp, source session, confidence, access frequency

Links create structure. When the user corrects the agent, the correction node links to the corrected node with a `supersedes` edge. When a decision leads to an outcome, the outcome links back with a `caused_by`

edge. When a preference is refined over multiple sessions, each refinement links to the previous with a `refines` edge.

The result is a directed graph of memories where traversal reveals not just what the agent knows, but how it came to know it — the full epistemic history of every piece of knowledge.

diagram

6.3 Context Chains

A context chain links execution contexts causally. Each execution context records a unit of work: what was done, why, what the outcome was, and what it led to.

Context chains answer questions that flat execution logs cannot:

- "Why did we restructure the DNS?" — Walk the chain backward from the DNS context to the domain registration context to the infrastructure discussion.
- "What happened after the PR was merged?" — Walk the chain forward from the merge context to the follow-up tasks.

- "What constraints apply to this task?" — Walk the chain of related contexts to find decisions that established constraints.

6.4 CTX Packages

Memory chains and context chains compile into **CTX packages** — pre-built, retrievable bundles stored in the database.

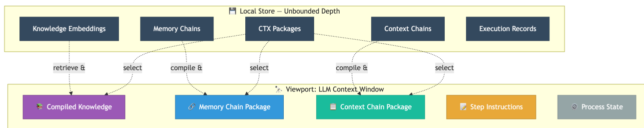
A CTX package is compiled from a set of chains, compressed into CTX format, and stored with metadata:

- **Source chains** — Which memory and context chains were compiled
- **Compiled size** — Token count of the compiled package
- **Raw size** — Token count of the uncompiled source material
- **Compression ratio** — Raw-to-compiled ratio
- **Freshness** — When the package was last recompiled
- **Access pattern** — How frequently the package is retrieved (for caching optimization)

Packages can be pre-compiled (for frequently accessed chains), on-demand (compiled at retrieval time), or auto-compiled (the runtime detects frequently co-retrieved chains and pre-compiles them as a package).

6.5 The Viewport Model

The context window is a viewport into the memory system:



diagram

The model sees 7,200 tokens of precision-compiled context. Behind that viewport sits a store containing the full history of every session the agent has ever run. The depth is effectively infinite — bounded only by local disk space, not by the context window.

6.6 Implications

The viewport model changes what is possible with a language model:

Perfect recall. The agent can retrieve and compile context from any previous session. A decision made six months ago is as accessible as one made six minutes ago.

No session boundaries. Memory chains span sessions continuously. The distinction between "this session" and "previous sessions" disappears — it is all one continuous memory, scoped through the viewport.

Accumulated expertise. Every correction, preference, and outcome is recorded. The agent's compiled context for a given task improves over time as more relevant memories accumulate. The agent gets better at your workflow because it remembers everything about your workflow.

Diagnostic capability. When the agent makes a mistake, the memory chain shows why — which memories informed the decision, which were missing, which were stale. This is debuggable, auditable intelligence.

✦ ✦ ✦

7. Composable Knowledge Packages

7.1 From Personal to Shared

The memory system described in Section 6 is personal by default — one user's memories, one user's chains, one user's machine. But compiled CTX packages are portable artifacts. They can be shared, composed, and distributed.

This transforms the CCR from a personal productivity tool into an organizational knowledge system.

7.2 Package Types

Personal knowledge packages. An individual's accumulated expertise in a domain — every decision, correction, pattern, and preference compiled into a retrievable bundle. "Everything I know about deploying to Kubernetes" as a CTX package for an engineer. "Everything I know about regulatory filings for Series B" for a startup lawyer. "Everything I know about patient intake workflows" for a clinic administrator. 3,000 tokens containing six months of accumulated context that would otherwise require reading hundreds of threads, documents, and emails.

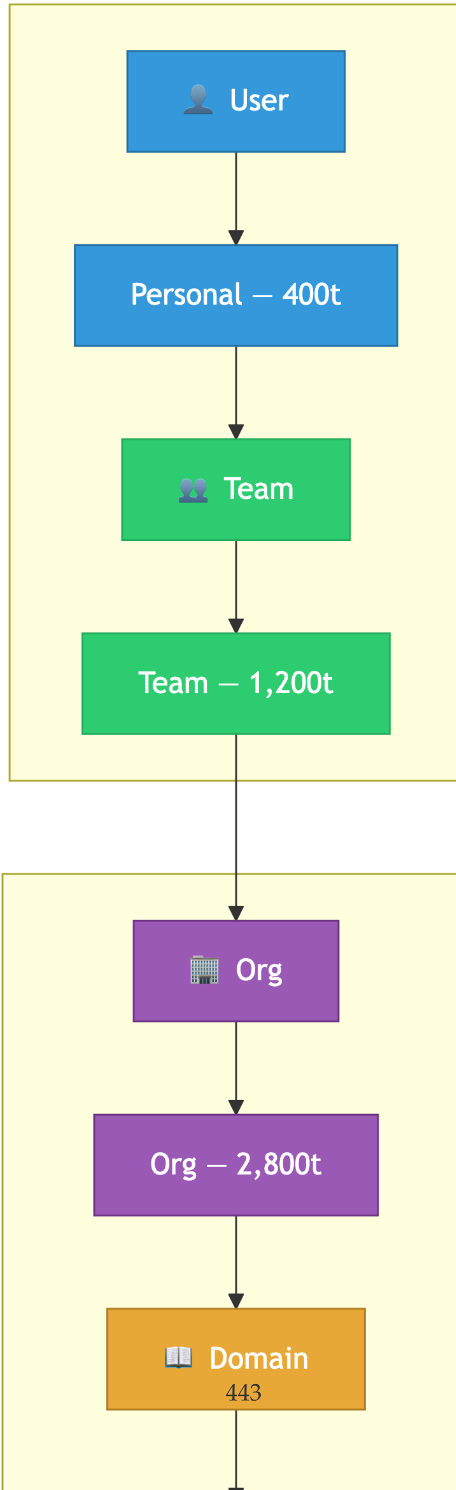
Team knowledge packages. A team's shared practices — standards, decisions, patterns, procedures — compiled from the merged memory chains of team members. New team members receive the team's

institutional knowledge as a compiled package. Their agent has the same context as a ten-year veteran on day one. This applies equally to an engineering team's architecture decisions, a sales team's qualification criteria, or a research group's methodology standards.

Organizational knowledge packages. An organization's tribal knowledge — the undocumented decisions, the unwritten rules, the historical context that explains why things work the way they do. Every organization has decades of accumulated knowledge that exists only in the heads of experienced people. When those people leave, the knowledge leaves with them. Compiled knowledge packages make tribal knowledge persistent, transferable, and precise.

Domain knowledge packages. Expertise in a specific domain — compiled from publications, documentation, best practices, and accumulated execution experience. "How to build event-driven architectures" or "SEC compliance for SaaS companies" or "Clinical trial protocol design" as a CTX package that any user's agent can consume.

7.3 Composition


User
Personal — 400t
Team
Team — 1,200t
Org
Org — 2,800t
Domain

diagram

Knowledge packages compose. A user's agent might load:

```
Active packages:
├── personal/my-preferences          (400 tokens)
├── team/backend-standards           (1,200
tokens)
├── org/architecture-decisions       (2,800
tokens)
├── domain/python-patterns           (1,500
tokens)
└── project/payment-service-context  (900 tokens)
                                     ────────────
                                     6,800 tokens
```

6,800 tokens carrying the combined expertise of the individual, the team, the organization, and the domain. A new hire's agent, on their first day, works with the same accumulated context as the most experienced person on the team — because the knowledge is compiled, not remembered.

7.4 Knowledge Models

At the limit, composed knowledge packages form a **local knowledge model** — a comprehensive, compiled representation of everything an individual or organization knows about their domain.

A knowledge model is not a language model. It does not generate text. It is a structured, indexed, compiled corpus that the language model consumes as

context. But it serves a similar function: it encodes expertise. The difference is that it encodes *specific* expertise — your architecture, your decisions, your patterns, your domain — rather than generic knowledge trained from internet text.

An experienced practitioner's knowledge model might contain:

- 50,000 memory nodes spanning two years of work
- 1,200 execution contexts recording every task completed
- 300 compiled CTX packages covering every project and domain they've touched
- 500,000 vector embeddings indexing their entire knowledge base

Compiled on demand, any subset of this knowledge model can be injected into an LLM call in under 10,000 tokens. The model works as if it has the practitioner's full expertise — because, through the viewport, it does.

7.5 Codifying Tribal Knowledge

Every organization has tribal knowledge — the accumulated, undocumented understanding that makes the system work. It lives in experienced people's heads, in hallway conversations, in threads

and documents that scroll off-screen. It is the most valuable knowledge the organization possesses and the least persistent.

The CCR codifies tribal knowledge structurally:

1. **Capture** — As people work with their agents, memory chains accumulate decisions, rationale, corrections, and context. The tribal knowledge that was previously ephemeral is now recorded as linked memory nodes.
2. **Compile** — Memory chains compile into knowledge packages. "Why the payment service uses eventual consistency" becomes a 600-token CTX package with the full decision chain, not a 5,000-word wiki page nobody reads.
3. **Share** — Knowledge packages are published to a team or organization knowledge store. Other users' agents consume them automatically when working in the relevant domain.
4. **Evolve** — As the system changes, new memory nodes extend the chains. Outdated knowledge is superseded by corrections. The packages recompile automatically. Tribal knowledge stays current because it is maintained by the same system that uses it.

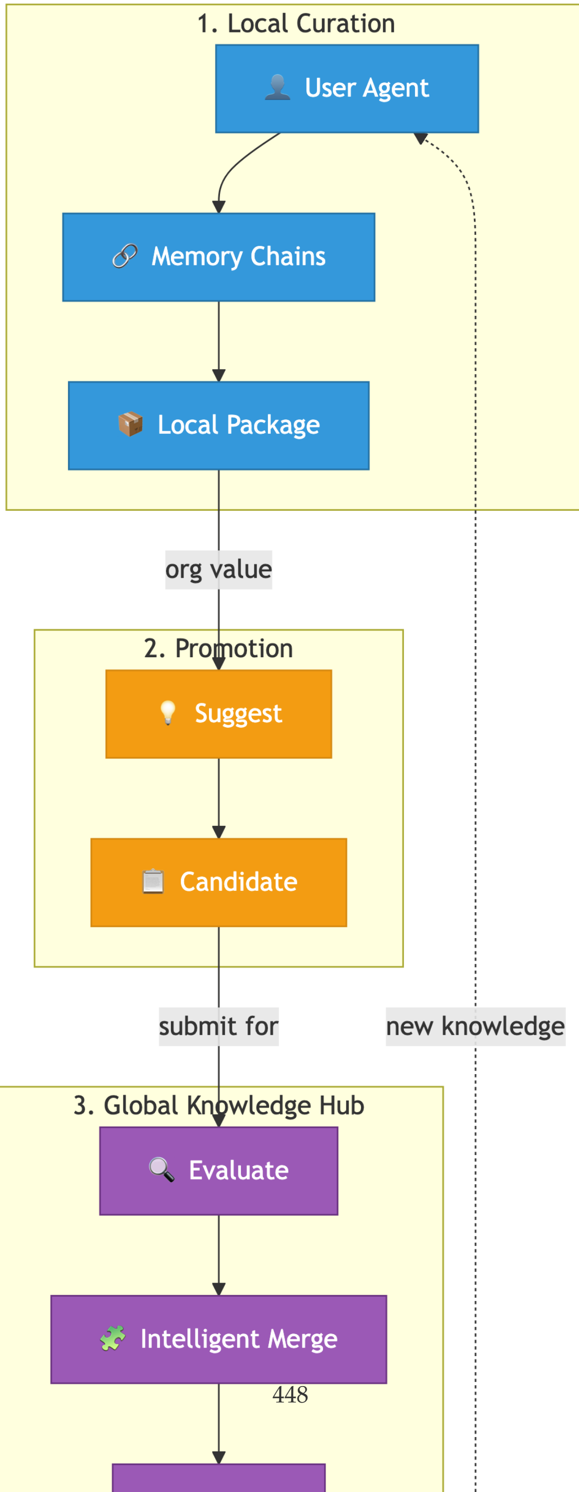
The result: tribal knowledge survives employee turnover. It survives team reorganizations. It survives the passage of time. The knowledge that used to walk

out the door when an experienced person left is now compiled, indexed, and available to every agent in the organization — permanently.

7.6 Knowledge Governance

The transition from personal knowledge to organizational knowledge requires governance — a structured pipeline for curating, promoting, evaluating, and distributing knowledge across an organization.

The governance pipeline:


1. Local Curation
User Agent
Memory Chains
Local Package
org value
2. Promotion
Suggest
Candidate
submit for
new knowledge
3. Global Knowledge Hub
Evaluate
Intelligent Merge

diagram

1. **Local curation** — Knowledge originates with individuals. Their agents accumulate memory chains and compile them into local knowledge packages. The user is the curator — they correct errors, refine context, and shape the knowledge through normal use. This is where knowledge quality is highest, because it is maintained by the person who uses it daily.
2. **Promotion** — When a user's local knowledge has organizational value — a decision that affects other teams, a pattern that applies across departments, a procedure that everyone should follow — the user (or their agent) suggests it for promotion. The package becomes a **candidate** for the organizational knowledge base.
3. **Evaluation at the hub** — A global knowledge hub receives candidates and evaluates them. This is not blind merging — the hub analyzes the candidate against the existing knowledge base, checks for conflicts with established decisions, validates that the knowledge is generalizable (not specific to one developer's environment), and assesses quality based on the underlying memory chains. Evaluation can be automated, human-reviewed, or a hybrid where the agent surfaces candidates for human approval.

4. **Intelligent merge** — Approved candidates are merged into the global knowledge base. "Intelligent" because the merge is not concatenation — it is structural integration. If the candidate extends an existing knowledge chain, it is linked. If it supersedes outdated knowledge, the old nodes are marked as superseded. If it conflicts with existing knowledge, the conflict is surfaced for resolution. The global knowledge base maintains the same chain structure as local packages — it is not a flat wiki, it is a compiled, linked, versioned corpus.
5. **Distribution** — Updated knowledge is pushed to all agents in the organization through the messaging backplane. The backplane is architecture-agnostic — it can be a local message bus for a small team, Apache Kafka for a large organization, or any pub/sub system in between. Agents subscribe to knowledge topics relevant to their current work. When the global hub publishes an update, subscribing agents receive the new compiled package and integrate it into their local knowledge store. The next time the agent needs that knowledge, it loads the latest version.

Backplane flexibility:

The messaging infrastructure scales with the organization:

Scale	Backplane	Pattern
Individual	Local filesystem	Direct read
Team (5-20)	Local message bus	Pub/sub, same network
Department (20-200)	Managed message queue	Topic-based routing
Enterprise (200+)	Kafka / cloud pub/sub	Partitioned, multi-region

The same knowledge governance pipeline works at every scale because the knowledge format is uniform (compiled CTX packages) and the distribution mechanism is pluggable. An organization starts with a local bus and migrates to Kafka as they grow — the knowledge packages, the governance pipeline, and the agent integration remain unchanged.

The governance loop:

Knowledge governance is not a one-time setup — it is a continuous loop. Local agents curate knowledge through daily use. Valuable knowledge is promoted. The hub evaluates and merges. Updated knowledge distributes to all agents. Those agents use the new knowledge, generating new memory chains, which produce new local packages, which may themselves be promoted. The organization's knowledge base is a living system that improves with every task every agent executes.

8. The Learning Loop

8.1 Process Discovery

The runtime does not only execute processes — it observes unstructured agent behavior and proposes new process definitions.

When the agent performs a sequence of actions outside of a defined process, the runtime records the sequence. If the same or similar sequence recurs across multiple sessions, the runtime proposes a process definition:

> *"This sequence has occurred 4 times with consistent steps and positive outcomes. Proposed process:* `fix_ci_failure` *(6 steps, 2 knowledge refs). Approve?"*

The proposal includes: - The proposed YAML definition - The execution history that inspired it - Confidence level based on repetition count, consistency of steps, and outcome quality

The user approves, modifies, or rejects. Approved proposals become versioned process definitions. The agent transitions from ad-hoc behavior to deterministic execution for that workflow.

8.2 Process Refinement

After a process has been executed multiple times, the runtime analyzes execution records and surfaces refinement suggestions:

- **Missing steps** — Actions the agent consistently takes after the process completes, suggesting the process definition is incomplete
- **Unnecessary steps** — Steps that are consistently skipped or produce no meaningful output
- **Missing gates** — Steps that frequently fail, suggesting a precondition that should be checked before execution
- **Missing knowledge** — Topics the model consistently requests mid-execution that weren't in the knowledge references
- **Redundant knowledge** — Knowledge references that don't correlate with improved outcomes

Each suggestion creates a proposed new version of the process. Approved suggestions increment the version. Rejected suggestions are recorded (to avoid re-suggesting).

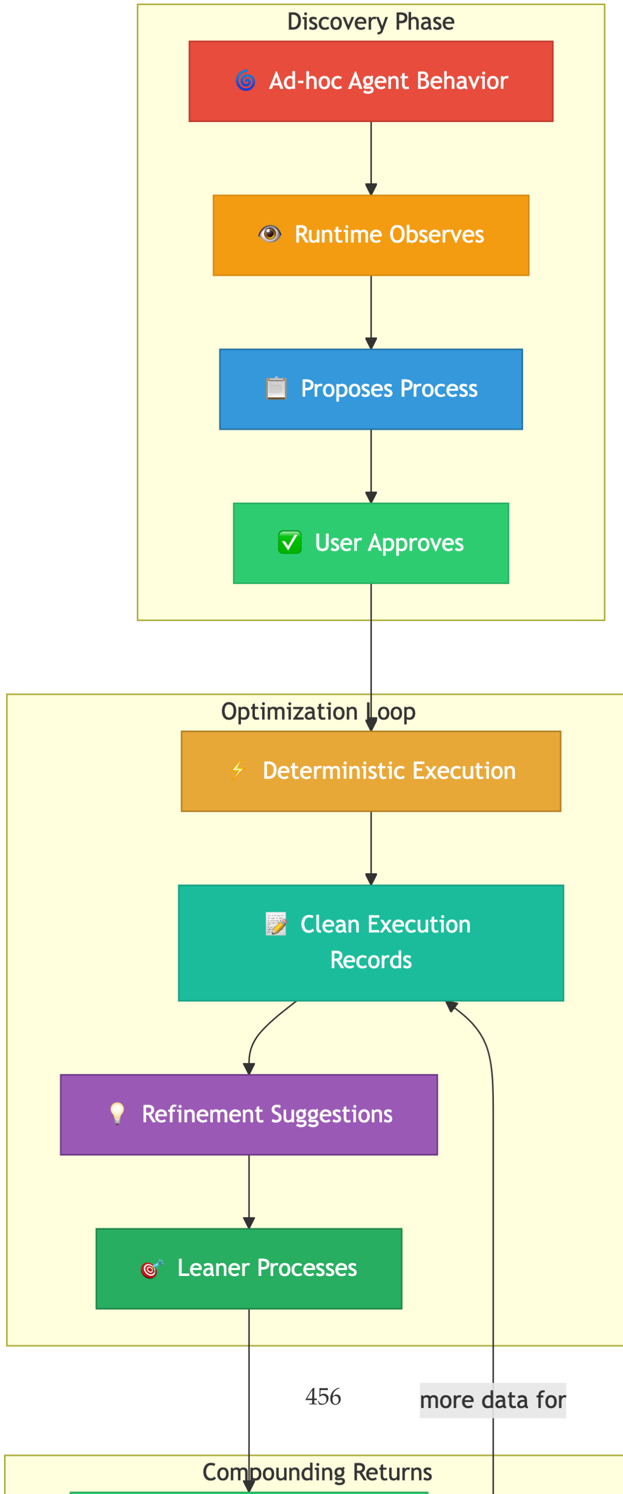
8.3 Context Optimization

The learning loop extends to context compilation. The runtime tracks which compiled context packages correlate with successful outcomes and which do not. Over time, this produces:

- **Leaner packages** — Removing knowledge that doesn't improve outcomes
- **Richer packages** — Adding knowledge that the model consistently needs but wasn't declared
- **Better scoping** — Narrowing or broadening per-step context based on observed usage patterns

The system gets cheaper to run the more you use it. Each execution provides data that the refinement loop uses to reduce waste in subsequent executions.

8.4 The Compound Effect


Discovery Phase
🌀 Ad-hoc Agent Behavior
👁️ Runtime Observes
📋 Proposes Process
✅ User Approves
Optimization Loop
⚡ Deterministic Execution
📝 Clean Execution Records
💡 Refinement Suggestions
🎯 Leaner Processes
more data for
Compounding Returns

diagram

Process discovery, process refinement, and context optimization compound:

1. The agent begins with no processes — all behavior is ad-hoc
2. The runtime observes repeated patterns and proposes processes
3. Processes replace ad-hoc behavior with deterministic execution
4. Deterministic execution produces cleaner execution records
5. Cleaner records enable more precise refinement suggestions
6. Refined processes use less context and fewer steps
7. Less context means fewer tokens per call
8. Fewer tokens means lower cost per execution
9. Lower cost enables more executions
10. More executions produce more data for further refinement

The system converges toward an optimum: maximum workflow reliability at minimum token cost, achieved through continuous, automated, user-approved refinement.

✦ ✦ ✦

9. Token Economics

9.1 The Cost Structure of Current Systems

LLM inference is priced per token. Input tokens (context) and output tokens (responses) each incur cost. For the purposes of this analysis, input tokens are the dominant cost driver — they are typically 3-10x more numerous than output tokens in agent workflows.

Current agent systems are structurally wasteful:

Waste Category	Description	Typical Overhead
Context stuffing	Full conversation history in every call	5-20x relevant content
Redundant retrieval	Same RAG passages injected repeatedly	2-5x per session
No scoping	All knowledge injected regardless of step	3-8x per step
No compression	Raw text, no semantic compression	1.4-2.5x compressible
Exploratory calls	Agent tries approaches, backtracks	2-4x deterministic path

These overheads multiply. A task that requires 5,000 tokens of relevant context might consume 200,000-500,000 tokens of input across a session of exploratory, unscoped, uncompressed calls.

9.2 The CCR Cost Structure

The Compiled Context Runtime eliminates each category of waste:

CCR Innovation	Waste Eliminated	Reduction
Process definitions	Exploratory calls	60-75% fewer calls
Per-step scoping	Context stuffing + no scoping	80-95% fewer tokens per call
CTX compilation	No compression	40-84% compression on remaining
Memory chains	Redundant retrieval + session loss	Near-zero redundancy

9.3 Quantitative Analysis

Per-task comparison:

Metric	Conventional Agent	CCR
Context per call	~50,000 tokens	~7,000 tokens
Calls per task	~20	~6
Total input tokens	~1,000,000	~42,000
Reduction	—	**96%**

The 96% figure reflects the compound effect of fewer calls (deterministic processes), smaller context per call (scoped + compiled), and no redundancy (chains eliminate re-retrieval).

Annual cost projections:

Scale	Conventional Cost/yr	CCR Cost/yr	Annual Savings
Solo practitioner	$2,400	$100	$2,300
10-person team	$24,000	$1,000	$23,000
100-person company	$240,000	$10,000	$230,000
1,000-person enterprise	$2,400,000	$100,000	$2,300,000
50,000-person Fortune 500	$120,000,000	$5,000,000	$115,000,000

Global projection:

LLM-assisted workflows extend far beyond software development. Analysts, researchers, writers, legal professionals, designers, consultants, educators, and administrators all use LLMs for knowledge work. The total addressable population is hundreds of millions of knowledge workers worldwide.

With conservative assumptions about adoption:

- 500 million knowledge workers globally (developers, analysts, researchers, writers, legal, consulting, education, etc.)
- 5% adoption rate: 25 million users
- Average savings of $2,300/year per user (solo-tier conservative)

- **$57.5 billion in annual savings globally**

At enterprise adoption rates with enterprise pricing, the figure is significantly higher. These are structural savings — they arise from architectural decisions, not from negotiating better API rates.

9.4 Beyond Cost: Reliability

Token reduction is not only an economic benefit. It directly improves model reliability.

A model processing 7,000 tokens of precision-compiled context attends more effectively than a model processing 50,000 tokens of raw, unscoped text. Attention dilution — the degradation of model performance as context grows — is a well-documented phenomenon. By reducing context to only what is relevant, the CCR improves not just cost but accuracy, consistency, and instruction-following.

The cheapest call is also the most reliable call. This is not a tradeoff — it is a structural advantage.

9.5 Beyond Cost: Energy and Environmental Impact

Token economics are not only a financial concern. Every token processed by a large language model requires GPU computation, which consumes electricity, which generates carbon emissions.

The energy cost of LLM inference is substantial and growing. A single GPU running inference consumes 300-700 watts. Data centers operating

thousands of GPUs for inference consume megawatts continuously. As LLM-assisted work scales to hundreds of millions of knowledge workers making hundreds of calls per day, the aggregate energy consumption becomes a material environmental concern.

The CCR's 96% reduction in input tokens translates directly to reduced computation:

- **Fewer tokens per call** — Less GPU time per inference. A 7,000-token input processes faster and consumes less energy than a 50,000-token input. The relationship is not linear — attention mechanisms scale quadratically with sequence length — so the energy savings from shorter contexts are superlinear.
- **Fewer calls per task** — Deterministic processes eliminate exploratory back-and-forth. Six calls instead of twenty means one-third the GPU invocations.
- **Compound reduction** — Fewer calls, each processing fewer tokens, each requiring less computation per token (due to quadratic attention scaling). The energy reduction compounds beyond the token reduction.

Projected energy savings at scale:

Scale	Conventional GPU-hours/yr	CCR GPU-hours/yr	Energy Saved
1,000-person enterprise	~175,000	~7,000	168,000 GPU-hours
Fortune 500 (50K users)	~8,750,000	~350,000	8,400,000 GPU-hours
Global (25M users at 5%)	~4,375,000,000	~175,000,000	4,200,000,000 GPU-hours

At approximately 500 watts per GPU, 4.2 billion GPU-hours represents **2,100 gigawatt-hours** of electricity saved annually — equivalent to powering roughly 190,000 American homes for a year.

The environmental case reinforces the economic case. Organizations adopting the CCR model reduce both their LLM spending and their computational carbon footprint. At global scale, the aggregate reduction in unnecessary GPU computation is measured in hundreds of gigawatt-hours — a meaningful contribution to sustainable AI infrastructure.

The impact extends beyond electricity. Large-scale GPU inference drives demand across the full data center supply chain:

- **Cooling** — GPUs generate heat proportional to computation. Data centers consume massive quantities of water and energy for cooling. Microsoft reported consuming 1.7 billion gallons of water in 2022, with AI workloads as a significant driver. Reducing unnecessary computation reduces cooling demand proportionally.
- **Hardware** — GPU manufacturing requires rare earth minerals, complex fabrication, and significant embodied carbon. Every unnecessary GPU deployed to handle wasteful inference is hardware that didn't need to be manufactured. Reducing demand for inference capacity reduces demand for GPU production.
- **Land and construction** — Data centers require physical space, power infrastructure, and network connectivity. The global data center construction boom is driven substantially by AI inference demand. Reducing that demand eases pressure on land, power grids, and construction resources.

- **Network** — Every API call transmits tokens across network infrastructure. Reducing token volume reduces network load, which reduces energy consumption at every hop between the user's machine and the inference cluster.

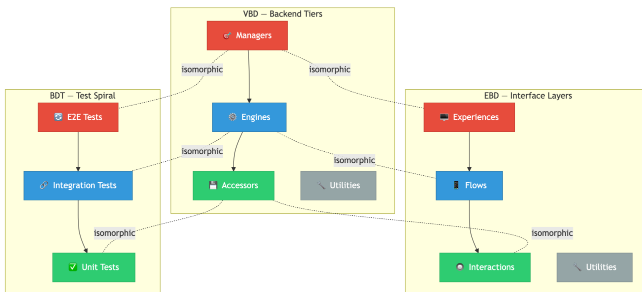
The CCR does not merely optimize a financial cost. It reduces the physical resource footprint of AI-assisted development at every layer of the infrastructure stack. The most sustainable token is the one that was never sent.

The most efficient inference call is the one that processes only what matters. The CCR ensures that every token that reaches the GPU earns its energy cost.

✦ ✦ ✦

10. Architectural Integration

10.1 Relationship to Harmonic Design



diagram

The Compiled Context Runtime is designed using Harmonic Design (HD) principles. The process engine, compilation pipeline, and memory system decompose into the standard HD tiers:

VBD — Backend Decomposition:

Component	Tier	Responsibility
ProcessManager	Manager	Matches triggers to processes, orchestrates execution
ProcessExecutionEngine	Engine	Runs steps, manages gates, records outcomes
ProcessDiscoveryEngine	Engine	Detects patterns in execution history, proposes processes
ProcessRefinementEngine	Engine	Analyzes outcomes, proposes improvements
CompilationEngine	Engine	CTX compilation pipeline
MemoryChainEngine	Engine	Chain traversal, linking, package compilation
ProcessDefinitionAccessor	Accessor	CRUD on process definitions (SQLite)
ExecutionRecordAccessor	Accessor	Read/write execution records (SQLite)
MemoryAccessor	Accessor	Read/write memory nodes and edges (SQLite)
KnowledgeStoreAccessor	Accessor	Vector similarity search, embedding management

EBD — Interface Decomposition:

Component	Layer	Responsibility
ProcessManagementExperience	Experience	Define, browse, and manage processes
ProcessExecutionFlow	Flow	Step-through execution with progress
ProcessSuggestionFlow	Flow	Review and approve suggestions
MemoryExplorerExperience	Experience	Browse and search memory chains
ChainDetailInteraction	Interaction	Inspect individual chain nodes and links

BDT — Test Spiral:

Scope	Coverage
Unit	Engines: step execution, gate evaluation, pattern detection, CTX compilation, chain traversal
Integration	Accessors with mocked SQLite/ vector DB; YAML parsing; compilation pipeline
E2E	Full trigger → match → gate → compile → inject → execute → record

10.2 Data Layer

All persistent state resides in two local stores:

SQLite — Process definitions, execution records, memory nodes, memory edges, context chain records, CTX package metadata, gate results, step outcomes.

Vector database — Knowledge embeddings, memory node embeddings, process description embeddings, execution summary embeddings. Used for similarity search during retrieval and for natural language queries ("find the process that handles CI failures").

Both stores are local files. No network dependency. No external service. Backup is a file copy.

✦ ✦ ✦

11. Validation and Falsifiability

11.1 Testable Claims

The CCR model makes specific, falsifiable claims:

1. **Token reduction:** Compiled, scoped context injection reduces input tokens per task by at least 80% compared to conventional context stuffing. Measurable by comparing total input tokens for identical tasks.
2. **Call reduction:** Deterministic process execution reduces the number of LLM calls per task by at least 50% compared to ad-hoc agent behavior. Measurable by counting calls for identical tasks.
3. **Outcome quality:** Models receiving precision-compiled context produce equal or better outcomes compared to models receiving raw, unscoped context. Measurable by blind evaluation of outputs.
4. **Memory accuracy:** Memory chains with typed links produce more accurate context retrieval than flat memory stores. Measurable by comparing retrieval precision and recall.
5. **Convergence:** The learning loop (discovery + refinement + context optimization) produces measurable improvements in token efficiency over time. Measurable by tracking tokens-per-task across process versions.

11.2 What Would Disprove the Model

The CCR model would be disproved if:

- Compiled context produces materially worse model outputs than raw context (compression is lossy in practice, not just in theory)
- Process definitions are too rigid to handle the variance of real-world tasks (deterministic steps cannot accommodate necessary creativity)
- The learning loop converges to local minima that are worse than ad-hoc behavior
- The overhead of compilation, retrieval, and chain management exceeds the savings from reduced tokens

These are empirical questions answerable through implementation and measurement.

✦ ✦ ✦

12. Conclusion

The Compiled Context Runtime is not an optimization applied to existing agent architecture. It is a different architecture. It replaces context stuffing with compiled injection, replaces prompt-dependent behavior with process-driven execution, and replaces session-bounded memory with persistent, linked, compilable chains.

The model's context window stops being a limitation and becomes an instrument. The agent stops forgetting and starts accumulating expertise. The cost of each execution drops as the system learns what context matters and what does not.

The system is local-first because the data it manages — workflows, memories, execution history, knowledge — is too valuable and too sensitive to externalize. It is open source because the structural advantages it provides should be accessible to everyone, not gated behind a platform subscription.

The economic impact is measured in tens of billions because the waste it eliminates is structural — embedded in how every current agent system is built. The Compiled Context Runtime does not ask users to write better prompts. It makes the prompt irrelevant as a vehicle for workflow definition, and makes the context window irrelevant as a constraint on memory depth.

What remains is the model doing what it does best — reasoning, creating, solving — with exactly the context it needs, compiled from everything the system has ever learned.

✦ ✦ ✦

Appendix A: Glossary

Attention Dilution — Degraded model performance caused by irrelevant tokens competing for attention in an oversized context window.

Build Primitive — The fourth execution primitive: creating new processes, knowledge artifacts, or tools when Learn identifies gaps.

Compiled Context — A precision-scoped, losslessly compressed package of knowledge and state injected into the model's context window for a specific process step.

Compiled Context Runtime (CCR) — An architectural model for agent execution that replaces context stuffing with compiled injection, prompt-dependent behavior with process-driven execution, and session-bounded memory with persistent chains.

Context Chain — A linked sequence of context records capturing the full history of a task's execution, compilable into a CTX package on demand.

Context Stuffing — The conventional approach of packing raw text into the context window before each inference call. The primary source of waste that CCR eliminates.

CTX Format — The lossless compression format used for compiled context packages, optimizing for token efficiency while preserving semantic completeness.

Execution Cycle — The five-primitive loop governing all agent work: Orchestrate → Execute → Learn → Build → Refine.

Execute Primitive — The second execution primitive: performing the actual work that produces external output.

Gate — A precondition declared in a process definition that must be satisfied before a step can proceed.

Knowledge Governance — The pipeline for curating, promoting, and distributing knowledge across organizational boundaries: local → team → organizational → hub.

Knowledge Package — A composable unit of domain knowledge with explicit scope, dependencies, and compilation rules.

Learn Primitive — The third execution primitive: analyzing outcomes at meta-learning (process improvement) and context-learning (domain knowledge) levels.

Local-First — The design principle that all agent data resides on the user's machine, with no workflow data crossing network boundaries except compiled context sent to the LLM.

Memory Chain — A persistent, linked sequence of memory records that accumulates across sessions, giving the model access to unbounded historical depth.

Model-Agnostic — The design property where intelligence accumulates in the data layer rather than model weights, making inference endpoints interchangeable.

Orchestrate Primitive — The first execution primitive: loading state, reading knowledge, compiling context, analyzing dependencies, and dispatching work.

Process Definition — A versioned, executable YAML specification of an agent workflow, declaring steps, gates, knowledge requirements, and trigger conditions.

Process Discovery — The system that detects repeated ad-hoc sequences and proposes new process definitions to codify them.

Refine Primitive — The fifth execution primitive: improving existing processes, knowledge, and tools based on execution analysis.

Token Economics — The quantitative analysis of cost reduction achieved by compiled context injection versus context stuffing, measured at individual, enterprise, and global scale.

Viewport — The conceptual model of the context window as a precision-scoped lens into a potentially unlimited local data store, rather than a hard size limit.

✦ ✦ ✦

References

William Christopher Anderson Anderson, W. C. *Volatility-Based Decomposition in Software Architecture: A Practitioner-Oriented Articulation.* Unpublished manuscript, 2026.

VBD provides the backend decomposition framework — Manager, Engine, Accessor, Utility tiers — that the CCR's process engine, compilation pipeline, and memory system are structured around. The volatility-driven tier assignments and communication rules described in this paper directly govern the CCR's component architecture.

Anderson, W. C. *Experience-Based Decomposition: A Practitioner-Oriented Articulation.* Unpublished manuscript, 2026.

EBD provides the interface decomposition framework — Experience, Flow, Interaction layers — that governs how users interact with the CCR through CLI, MCP tools, and future interfaces. The separation of orchestration from interaction mirrors the CCR's own separation of process management from step execution.

Anderson, W. C. *Boundary-Driven Testing: A Practitioner-Oriented Articulation.* Unpublished manuscript, 2026.

BDT provides the test architecture — unit, integration, and end-to-end spirals mirroring component tiers — that validates the CCR's

boundaries. The structural isomorphism between component tiers and test scopes ensures that each boundary in the system has a corresponding test boundary.

Anderson, W. C. *Harmonic Design: A Unified Software Engineering Practice.* Unpublished manuscript, 2026.

Harmonic Design unifies VBD, EBD, and BDT as harmonics of the same fundamental principle: organize by anticipated change. The CCR is built as an HD system — its backend decomposes by VBD, its interfaces by EBD, its tests by BDT, and the three frameworks reinforce each other structurally. The CCR's own knowledge governance, process definitions, and compilation pipeline are all governed by HD principles.

David Lorge Parnas Parnas, David L. "On the Criteria to Be Used in Decomposing Systems into Modules." *Communications of the ACM*, vol. 15, no. 12, 1972, pp. 1053–1058.

Parnas's foundational insight — that systems should be decomposed by what is likely to change, not by workflow or data flow — is the intellectual ancestor of VBD and, by extension, the CCR's own decomposition. The CCR's separation of process definitions (highly volatile) from the compilation pipeline (moderately volatile) from the storage layer (stable) directly reflects Parnas's criteria.

Juval Lowy Lowy, Juval. *Righting Software.* Addison-Wesley, 2019.

Lowy's IDesign methodology originated the volatility-based decomposition approach, the Manager/Engine/Accessor/Utility taxonomy, and the communication rules that VBD articulates. The CCR's architectural structure — managers orchestrating engines that encapsulate logic over accessors that isolate external resources — is a direct application of Lowy's system.

Martin Fowler Fowler, Martin. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2002.

Fowler's patterns for layered architecture, repository abstraction, and unit of work inform the CCR's accessor patterns and state management. The SynapseAccessor and VectorAccessor patterns in the CCR follow Fowler's repository pattern adapted for filesystem and vector database access.

Eric Evans Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley, 2003.

Evans's bounded contexts inform the CCR's knowledge package boundaries. Each knowledge package — personal, team, organizational, domain — functions as a bounded context with explicit interfaces

for composition. The CCR's knowledge governance pipeline reflects DDD's strategic design principles applied to knowledge management rather than code.

Ashish Vaswani et al. Vaswani, Ashish, et al. "Attention Is All You Need." *Advances in Neural Information Processing Systems*, 2017.

The transformer architecture's quadratic attention scaling with sequence length is the fundamental constraint that makes context compilation economically valuable. The CCR's token economics — superlinear energy savings from shorter contexts — derive directly from the attention mechanism's computational characteristics.

Nelson F. Liu et al. Liu, Nelson F., et al. "Lost in the Middle: How Language Models Use Long Contexts." *Transactions of the Association for Computational Linguistics*, 2024.

Liu et al.'s demonstration that language models attend poorly to information in the middle of long contexts provides empirical support for the CCR's compilation approach. By delivering only relevant, precision-scoped context rather than large volumes of raw text, the CCR avoids the "lost in the middle" phenomenon entirely.

✦ ✦ ✦

Author's Note

The Compiled Context Runtime synthesizes ideas from multiple domains: process engineering, knowledge management, compiler design, and agent architecture. The architectural framework — Harmonic Design and its constituent practices — originates from the author's prior work articulating VBD, EBD, BDT, and HD. The specific application of these frameworks to agent runtime architecture, compiled context injection, memory chains, composable knowledge packages, knowledge governance, and dynamic model selection is, to the author's knowledge, novel.

The system described in this paper is not theoretical. The author has built and operates a working implementation of the core concepts: process definitions in YAML governing agent execution, a knowledge index with compiled context injection per step, memory that persists across sessions and accumulates over months, execution contexts that track every task from trigger to completion, and a knowledge governance pipeline that curates and distributes knowledge across agent sessions. The token economics are derived from measured reductions in actual agent workflows, not projections from hypothetical systems.

The decision to scope the CCR to all knowledge workers — not just software developers — reflects the observation that every LLM-assisted workflow, regardless of domain, suffers from the same structural waste: bloated context, stateless execution, no learning between sessions, and no process discipline. A lawyer reviewing contracts, a researcher analyzing papers, an analyst building financial models, and a developer writing code all benefit equally from compiled context, deterministic processes, and accumulated memory. The architecture is domain-agnostic because the problem it solves is domain-agnostic.

The model-agnostic design — where the runtime dynamically selects the optimal model per step based on task requirements and available capabilities — is a deliberate architectural choice, not a compatibility feature. Intelligence should accumulate in the data layer (processes, memories, knowledge), not in any particular model's weights. When the data layer carries the intelligence, models become interchangeable inference endpoints, and organizations are freed from vendor lock-in. The knowledge you build today works with whatever model exists tomorrow.

The knowledge governance pipeline — local curation, organizational promotion, hub evaluation, intelligent merge, and backplane distribution — addresses what the author considers the most valuable

application of the CCR: codifying tribal knowledge. Every organization loses critical knowledge when experienced people leave. The CCR makes that knowledge persistent, compilable, and distributable. At organizational scale, this is not a productivity optimization — it is a structural solution to institutional knowledge loss.

✦ ✦ ✦

Distribution Note

This document is provided for informational and educational purposes. It may be shared internally within organizations, used as a reference in architectural and design discussions, or adapted for non-commercial educational use with appropriate attribution. All examples are generalized and abstracted to avoid disclosure of proprietary or sensitive information.

✦ ✦ ✦

Copyright (c) 2026 William Christopher Anderson. All rights reserved.

APPENDIX G

Swarm Architecture

Swarm Architecture

Bounded Parallel Agent Execution

Author: William Christopher Anderson **Date:** April 2026 **Version:** 1.0

✦ ✦ ✦

Executive Summary

The Compiled Context Runtime (CCR) solves the problem of agent continuity. Process definitions codify what to do. Compiled context injection provides what to know. Memory chains preserve what was learned. Together, they produce an agent that operates with precision, consistency, and accumulating intelligence.

But the CCR, as described in its foundational whitepaper, operates as a single sequential executor. One agent. One process. One step at a time. This is correct for most work — the majority of knowledge tasks are inherently sequential, and parallelism introduces coordination complexity that rarely justifies the overhead.

Some work, however, is naturally parallel. A code review that spans twelve files can be decomposed into twelve independent analyses. A migration that touches eight databases can execute eight schema changes simultaneously. A research task that consults fourteen sources can dispatch fourteen retrieval operations and merge the results. In these cases, sequential execution is not merely slow — it is structurally wrong. The task's natural shape is parallel, and forcing it through a sequential pipeline distorts the work.

Swarm Architecture extends the CCR with a model for bounded parallel agent execution. It introduces three constructs:

1. **Swarms** — bounded groups of agents executing task instances in parallel, governed by a single coordinator and a single correlation identity. A swarm is not a cluster, not a pool, not an unbounded collection of workers. It is a precisely scoped execution boundary: one process step

decides to fan out, the swarm executes the parallel work, and the results converge before execution continues.

2. **Containment rules** — structural constraints that prevent swarm workers from escaping their execution boundary. A swarm worker may spawn sub-agents within its own process boundary, but it may not join another swarm, initiate new swarms, or communicate laterally with sibling workers. These rules are not conventions. They are enforced by the runtime.
3. **Convergence protocols** — mechanisms for collecting, merging, and validating the results of parallel execution before the parent process continues. Convergence is not implicit. It is a defined step in the process, with explicit merge strategies, conflict resolution rules, and quality gates.

A fourth property emerges from the containment model that is not obvious from its safety-oriented design:

1. **Location independence** — because containment rules prohibit workers from accessing shared state, communicating laterally, or reaching outside their execution boundary, workers have no requirement for co-location. A worker can execute on the coordinator's machine, on a cloud

> instance, on an edge device in a factory, or on a partner organization's infrastructure across the planet. The containment rules designed for safety become the enabling constraints for distribution. The compiled context boundary becomes the security boundary — workers receive only what their task requires, and cannot leak what they were never given.

This property transforms the scope of what the architecture can do. The same swarm model that parallelizes a local code review can distribute a compliance analysis across twelve jurisdictions, a research synthesis across institutions that cannot share raw data, or a global monitoring operation across edge devices on every continent. The mechanism is identical. The topology varies.

✦ ✦ ✦

Abstract

The Compiled Context Runtime provides process-driven, context-compiled agent execution with persistent memory. Its sequential execution model is correct for the majority of agent workflows, but structurally inadequate for tasks whose natural decomposition is parallel. Swarm Architecture extends the CCR with bounded parallel execution: swarms of agents that execute independent task instances

simultaneously under a single coordinator, governed by containment rules that prevent execution boundary violations, and converged through explicit merge protocols before the parent process continues. This paper describes the architectural model, the containment and coordination mechanisms, the convergence protocols, the relationship to the CCR's process definition language, the failure and recovery model, the cost implications of parallel versus sequential execution, and the distributed execution model that emerges from the containment architecture — enabling swarms whose workers span local machines, cloud regions, edge devices, and partner infrastructure without sacrificing determinism, auditability, or data security.

✦ ✦ ✦

1. Introduction

1.1 The Sequential Assumption

The Compiled Context Runtime, as described in its foundational paper, executes processes as ordered sequences of steps. Step one completes before step two begins. Each step receives compiled context scoped to its requirements. Each step's output is captured in execution history and available to subsequent steps. The model is simple, predictable, and auditable.

This sequential model is not a limitation — it is a design choice rooted in a structural observation: most knowledge work is inherently sequential. Writing code requires understanding the context before writing. Reviewing a pull request requires reading the changes before forming an opinion. Planning a project requires understanding the dependencies before sequencing the work. Forcing parallelism onto inherently sequential work produces coordination overhead without meaningful speedup.

But not all work is sequential.

1.2 The Parallelism That Already Exists

Consider a process that reviews a large pull request. The process definition might specify:

1. Retrieve the PR metadata and changed file list
2. For each changed file, analyze the diff against the relevant architectural standards
3. Synthesize individual file analyses into a coherent review
4. Post the review

Steps 1, 3, and 4 are inherently sequential. Step 2 is inherently parallel — the analysis of `billing_engine.py` does not depend on the analysis of `payment_accessor.py`. They share no state. They require no coordination. They can execute simultaneously without affecting each other's results.

Today, the CCR executes step 2 as a loop: analyze file one, then file two, then file three. Each analysis is independent, but they execute sequentially because the runtime has no mechanism to express or execute parallel work. The result is correct but slow — a twelve-file review takes twelve sequential analysis cycles instead of one parallel cycle.

This is not a theoretical concern. It is a concrete performance penalty applied to every naturally parallel task the agent encounters.

1.3 Why Not General Multi-Agent Systems?

The obvious response is: deploy multiple agents. Let them coordinate. Let them discover work, distribute it, and merge results dynamically.

This is the approach taken by most multi-agent frameworks, and it fails for predictable reasons.

Containment failure. When agents can spawn other agents without structural constraints, the system's execution boundary becomes unbounded. An agent debugging a test failure spawns an agent to read the source code, which spawns an agent to check the git history, which spawns an agent to analyze the CI configuration. Each spawn is locally reasonable. The aggregate is an uncontrolled expansion of execution scope, token consumption, and coordination complexity.

Coordination overhead. General multi-agent coordination requires consensus mechanisms, shared state management, conflict resolution, and deadlock detection. These mechanisms are well-understood in distributed systems, but they introduce complexity that is disproportionate to the problem. The CCR's value proposition is deterministic, auditable execution. Adding distributed coordination undermines that proposition.

Emergent behavior. When multiple agents operate with overlapping scope and lateral communication, the system's behavior becomes emergent rather than specified. The process definition says what should happen; the agents decide what actually happens. This is the opposite of the CCR's design philosophy, where the process definition is the single source of truth for execution.

Swarm Architecture avoids all three failure modes by constraining parallelism to a specific, bounded pattern: fan out, execute independently, converge. No lateral communication. No dynamic scope expansion. No emergent coordination.

1.4 Scope of This Paper

This paper describes Swarm Architecture as an extension to the Compiled Context Runtime. It assumes familiarity with the CCR's process definitions, compiled context injection, memory

chains, and execution model. Readers unfamiliar with these concepts should consult the CCR whitepaper before proceeding.

The paper covers the swarm execution model, containment rules, convergence protocols, process definition extensions, failure and recovery, cost analysis, and the relationship to VBD component architecture. It does not cover general-purpose multi-agent orchestration, distributed consensus algorithms, or agent-to-agent communication protocols — these are explicitly out of scope.

✦ ✦ ✦

2. The Swarm Model

2.1 Definition

A **swarm** is a bounded group of agents executing independent task instances in parallel, governed by a single coordinator, identified by a single correlation ID, and converged through an explicit merge step before the parent process continues.

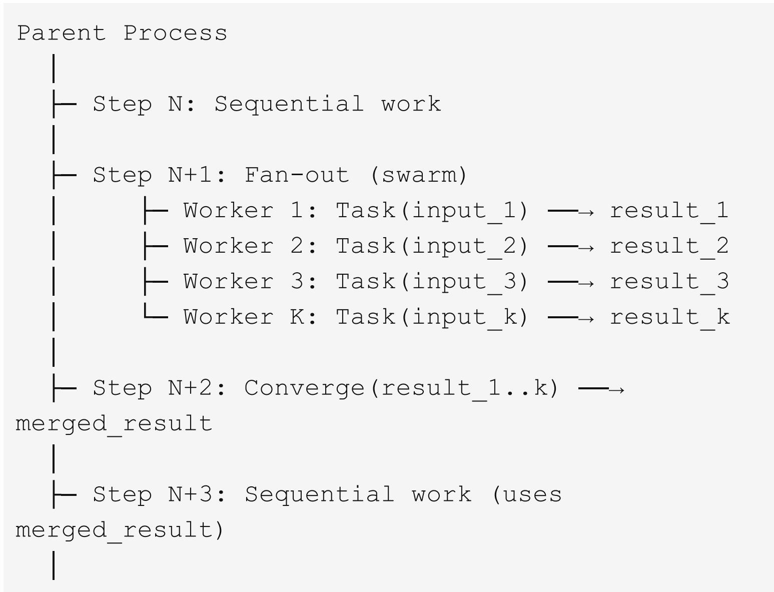
Every swarm has exactly five properties:

1. **A parent step** — the process step that initiated the fan-out. The parent step is suspended until convergence completes.

2. **A task blueprint** — the process definition (or task definition) that each worker executes. All workers in a swarm execute the same blueprint against different inputs.
3. **An input set** — the collection of independent work items to be processed. Each item becomes the input to one worker instance.
4. **A convergence strategy** — the mechanism for collecting, merging, and validating worker outputs before returning control to the parent process.
5. **A correlation ID** — a unique identifier that links every worker instance, every log entry, every memory record, and every artifact produced by the swarm back to the parent step that initiated it.

2.2 The Fan-Out / Converge Pattern

Swarm execution follows a single pattern:



```
Parent Process
   │
   ├─ Step N: Sequential work
   │
   ├─ Step N+1: Fan-out (swarm)
   │        ├─ Worker 1: Task(input_1) ──→ result_1
   │        ├─ Worker 2: Task(input_2) ──→ result_2
   │        ├─ Worker 3: Task(input_3) ──→ result_3
   │        └─ Worker K: Task(input_k) ──→ result_k
   │
   ├─ Step N+2: Converge(result_1..k) ──→
merged_result
   │
   ├─ Step N+3: Sequential work (uses
merged_result)
   │
```

The pattern is deliberately simple. There is no nesting of swarms within swarms. There is no lateral communication between workers. There is no dynamic addition of work items after fan-out begins. The swarm is a structural primitive — a single level of parallelism — not a recursive coordination framework.

2.3 What a Swarm Is Not

A swarm is not a thread pool. Thread pools are infrastructure-level concurrency mechanisms. Swarms are architecture-level execution patterns. A swarm might be implemented using threads, processes, API calls, or distributed workers — the implementation is invisible to the process definition.

A swarm is not a MapReduce job. MapReduce operates on data partitions with a fixed reduce function. Swarms operate on task instances with configurable convergence strategies. The workers are agents executing process steps, not functions applied to data shards.

A swarm is not an agent swarm in the multi-agent literature. The term "swarm" in multi-agent systems typically implies emergent coordination, stigmergic communication, and self-organizing behavior. None of these properties apply here. A CCR swarm is deterministic, bounded, and fully specified by the process definition. The term is used for its intuitive meaning — a group working in parallel — not for its academic connotations.

✦ ✦ ✦

3. Containment

3.1 The Containment Problem

Parallelism without containment is the defining failure mode of multi-agent systems. When an agent can spawn other agents without constraint, three problems emerge:

1. **Scope creep** — Each spawned agent may itself spawn agents, producing an expanding tree of execution that no single process definition governs.
2. **Resource exhaustion** — Each agent consumes context window tokens, API calls, and memory. Unbounded spawning produces unbounded cost.
3. **Audit failure** — When the execution tree is dynamic and unbounded, tracing what happened and why becomes intractable.

Swarm Architecture prevents all three through structural containment rules enforced by the runtime.

3.2 The Three Containment Rules

Rule 1: A swarm worker may not join another swarm.

A worker is executing a task instance within a specific swarm boundary. It may not register itself as a worker in a different swarm, even if that swarm is

executing the same task blueprint. This prevents cross-swarm contamination and ensures that each swarm's execution boundary is closed.

Rule 2: A swarm worker may not initiate a new swarm.

If a worker's task requires further parallelism, it must express that need through its process definition, and the parent process must orchestrate it as a separate swarm step. Workers do not have the authority to create swarms. Only the process coordinator does. This prevents recursive fan-out and bounds the total parallelism to what the process definition explicitly specifies.

Rule 3: A swarm worker may not communicate laterally with sibling workers.

Workers in the same swarm share a correlation ID, but they do not share state, messages, or coordination signals. Worker 3 cannot read Worker 7's intermediate results. Worker 7 cannot signal Worker 3 to change its approach. The only communication path is vertical: worker to coordinator (via result submission) and coordinator to worker (via task input and compiled context).

3.3 What Workers Can Do

The containment rules constrain inter-swarm and inter-worker behavior. Within its own execution boundary, a worker has full CCR capabilities:

- **Execute process steps** — The worker runs its assigned task blueprint as a normal CCR process.
- **Use compiled context** — The worker receives context compiled for its specific input, just as any CCR process step would.
- **Record memory** — The worker writes to memory chains, tagged with the swarm's correlation ID.
- **Spawn sub-agents** — The worker may use the CCR's standard sub-agent mechanism (tool calls, delegate steps) within its own process boundary. These sub-agents are scoped to the worker's process and do not constitute a new swarm.
- **Produce artifacts** — The worker generates output artifacts that are collected during convergence.

The distinction is precise: a worker is a full CCR agent within its boundary, but it cannot extend its boundary or interact with agents outside it.

3.4 Runtime Enforcement

Containment rules are not guidelines. They are enforced by the runtime through structural checks:

- **Swarm registration** — When a swarm is created, each worker receives a swarm scope token. API calls that would create or join a swarm are rejected if the calling context already holds a swarm scope token.
- **Communication isolation** — Workers receive isolated memory chain namespaces. Cross-worker memory queries are structurally impossible because the namespace scoping prevents it.
- **Execution boundary tracking** — The runtime maintains an execution tree with strict parent-child relationships. Any attempt to create a lateral edge (worker-to-worker) or an upward edge (worker initiating a new swarm) is rejected.

✦ ✦ ✦

4. Convergence

4.1 The Convergence Step

When all workers in a swarm complete (or when a timeout or failure threshold is reached), the swarm enters convergence. Convergence is not implicit — it is a defined step in the parent process with its own context, logic, and quality gates.

The convergence step receives:

- The ordered list of worker results
- The original input set (for correlation)
- Metadata about each worker's execution (duration, token usage, success/failure status)
- The swarm's correlation ID (for memory chain queries)

The convergence step produces:

- A merged result that the parent process uses in subsequent steps
- A convergence report (which workers succeeded, which failed, what conflicts were resolved)
- Memory chain entries recording the swarm's execution for future reference

4.2 Convergence Strategies

The process definition specifies which convergence strategy applies. The CCR provides four built-in strategies and supports custom strategies:

Collect — The simplest strategy. Worker results are collected into an ordered list and passed to the next step without transformation. The parent process is responsible for interpretation. Appropriate when results are independent observations that don't need merging (e.g., file-level code review comments).

Merge — Worker results are combined into a single output using a merge function specified in the process definition. Conflicts are resolved by the merge function. Appropriate when results contribute to a single deliverable (e.g., parallel document sections assembled into a complete document).

Vote — Worker results are treated as votes. The convergence step tallies results and selects the majority or highest-confidence output. Appropriate when multiple workers analyze the same input from different perspectives and the system needs a consensus decision (e.g., parallel classification with confidence scoring).

Reduce — Worker results are processed sequentially through a reduction function, producing a single accumulated result. Appropriate when results

need ordered integration (e.g., parallel test results reduced into a pass/fail summary with aggregated metrics).

Custom — The process definition specifies a convergence process (itself a CCR process definition) that receives the worker results and produces the merged output. This allows arbitrary convergence logic, including multi-step convergence with its own compiled context and quality gates.

4.3 Partial Convergence and Failure Thresholds

Not all workers may succeed. A swarm of twelve workers analyzing twelve files may have one worker fail due to a token limit, a model error, or an input that cannot be processed. The convergence strategy must handle partial results.

The process definition specifies failure behavior through two parameters:

- `min_success_ratio` — The minimum fraction of workers that must succeed for convergence to proceed. Default is 1.0 (all workers must succeed). Setting this to 0.8 means convergence proceeds if at least 80% of workers succeed; failed workers' inputs are recorded for retry or manual review.

- **`failure_action`** — What happens when a worker fails: `retry` (resubmit the failed input to a new worker), `skip` (proceed without the failed input), `abort` (fail the entire swarm and return control to the parent process's error handler).

These parameters allow the process definition to express the task's tolerance for partial results without embedding retry logic in every worker.

✦ ✦ ✦

5. Process Definition Extensions

5.1 Swarm-Eligible Steps

A process step becomes swarm-eligible through a `fan_out` declaration in the process definition:

```yaml
process: review_pull_request
version: 1

steps:
  - name: retrieve_pr
    action: "Fetch PR metadata and changed file
        list"
    output: pr_files

  - name: analyze_files
    action: "Analyze each changed file against
        architectural standards"
    fan_out:
      over: pr_files
      task: analyze_single_file
      concurrency: 8
      convergence:
        strategy: collect
        min_success_ratio: 0.9
        failure_action: skip
    output: file_analyses

  - name: synthesize_review
    action: "Combine file analyses into a
        coherent review"
    input: file_analyses
    output: review

  - name: post_review
    action: "Post the review to the pull request"
    input: review
```

The `fan_out` block specifies:

- **`over`** — The collection to iterate. Each element becomes the input to one worker.

- `task` — The task blueprint each worker executes. This is a reference to a task definition.
- `concurrency` — The maximum number of workers executing simultaneously. This is a resource constraint, not a parallelism constraint — all items will be processed, but at most `concurrency` workers run at any time.
- `convergence` — The convergence strategy and failure parameters.

5.2 Task Blueprints

A task blueprint is a process definition designed to be executed by a swarm worker. It is a standard CCR process with one constraint: it must accept a single input item and produce a single output result.

```yaml
task: analyze_single_file
version: 1

knowledge:
  - architecture/patterns/vbd-component-taxonomy
  - coding/python/style-guide

input:
  file_path: string
  diff_content: string
  pr_context: string

steps:
  - name: analyze
    action: "Analyze the diff against VBD
        standards and coding conventions"
    output: analysis

  - name: format
    action: "Format the analysis as a structured
        review comment"
    input: analysis
    output: review_comment

output: review_comment
```

Task blueprints inherit the full CCR capability set: compiled context injection, knowledge references, gates, and memory recording. They are not reduced-capability processes — they are full processes executing within a containment boundary.

5.3 Behavior Hints

Process and task definitions may include behavior hints that inform the runtime's scheduling and resource allocation decisions:

```yaml
behavior:
  swarm_eligible: true
  containment: strict
  event_topics:
    - task.lifecycle
    - artifact.produced
  estimated_duration: short
  model_requirements:
    reasoning_depth: moderate
    code_generation: true
```

Behavior hints are advisory, not prescriptive. The runtime uses them for optimization — routing short tasks to faster models, pre-allocating resources for large fan-outs, selecting appropriate event channels — but the process definition's semantic meaning does not depend on them.

✦ ✦ ✦

6. Coordination

6.1 Correlation IDs

Every entity created during a swarm's execution — worker instances, memory records, artifacts, log entries, execution records — carries the swarm's correlation ID. This produces a complete, traceable execution graph:

```
Swarm: swarm-a1b2c3d4
  ├─ Worker: swarm-a1b2c3d4/worker-001
  │    ├─ Memory: mem-xxx (chain: review, corr:
swarm-a1b2c3d4)
  │    └─ Artifact: art-xxx (corr: swarm-
a1b2c3d4)
  ├─ Worker: swarm-a1b2c3d4/worker-002
  │    ├─ Memory: mem-yyy (chain: review, corr:
swarm-a1b2c3d4)
  │    └─ Artifact: art-yyy (corr: swarm-
a1b2c3d4)
  └─ Convergence: swarm-a1b2c3d4/converge
       └─ Artifact: art-zzz (merged result, corr:
swarm-a1b2c3d4)
```

Correlation IDs enable three capabilities:

1. **Audit** — Given a swarm's correlation ID, the runtime can reconstruct the complete execution history: which workers ran, what each produced, how results were merged, what the final output was.

2. **Cost attribution** — Token usage, API calls, and execution time are attributed to the swarm and, through the correlation ID, to the parent process step that initiated it.
3. **Memory scoping** — Memory chain queries can be scoped to a swarm's correlation ID, allowing the convergence step to access the collective observations of all workers without pollution from unrelated memory.

6.2 Event-Driven Lifecycle

Swarm lifecycle events are published to event topics, allowing the parent process, monitoring systems, and the learning loop to observe swarm execution without polling:

Event	Payload	Published When
`swarm.created`	Swarm ID, parent step, input count	Fan-out initiated
`worker.started`	Worker ID, input item	Worker begins execution
`worker.completed`	Worker ID, result summary, token usage	Worker finishes successfully
`worker.failed`	Worker ID, error details, retry eligible	Worker encounters an error
`swarm.converging`	Swarm ID, success count, failure count	All workers done, convergence starting
`swarm.completed`	Swarm ID, merged result summary, total cost	Convergence complete
`swarm.aborted`	Swarm ID, reason, partial results	Failure threshold exceeded

Events are published to the topics declared in the task blueprint's `behavior.event_topics`. The parent process may subscribe to these events for progress reporting, but the events are informational — they do not affect execution flow.

✦ ✦ ✦

7. Failure and Recovery

7.1 Worker Failure Modes

Swarm workers can fail in three categories:

Transient failures — API rate limits, network timeouts, model overload. These are retryable. The runtime resubmits the failed input to a new worker instance, up to a configurable retry limit.

Input failures — The input item is malformed, too large for the context window, or references content that doesn't exist. These are not retryable with the same input. The convergence strategy's `failure_action` determines the response: skip the item, abort the swarm, or flag for manual review.

Structural failures — The task blueprint itself is flawed: a step references nonexistent knowledge, a gate condition is unsatisfiable, or the output schema doesn't match the convergence strategy's expectations.

These indicate a process definition error and always abort the swarm. The error is recorded in the execution history for the learning loop to analyze.

7.2 Swarm-Level Recovery

When a swarm is aborted, the parent process's error handler receives:

- The partial results from workers that succeeded
- The error details from workers that failed
- The swarm's execution metadata (duration, token usage, worker count)

The parent process may retry the entire swarm, proceed with partial results, fall back to sequential execution, or escalate to the user. The decision logic is expressed in the process definition's error handling steps — not in the swarm infrastructure.

7.3 Idempotency Requirement

Task blueprints used in swarms must be idempotent — executing the same input twice must produce the same result without side effects. This is required because the retry mechanism may resubmit inputs, and the system must guarantee that retried work does not corrupt state.

In practice, this means swarm tasks should: - Read from compiled context and input parameters only - Write to memory chains (which are append-only and

thus naturally idempotent) - Produce artifacts as output (which are captured by the convergence step, not written to external systems) - Defer side effects (file writes, API calls, notifications) to the parent process's post-convergence steps

✦ ✦ ✦

8. Cost Model

8.1 Sequential Baseline

For a task with N independent items, sequential execution requires:

- N × (context compilation cost + inference cost + memory recording cost)
- Total wall-clock time: N × average_step_duration
- Total tokens: N × average_tokens_per_step

8.2 Swarm Execution

The same task with swarm execution requires:

- N × (context compilation cost + inference cost + memory recording cost) — the total token cost is identical
- 1 × convergence cost — an additional inference call to merge results

- Total wall-clock time: max(worker_durations) + convergence_duration ≈ average_step_duration + convergence_duration

The critical insight: **swarms do not reduce token cost. They reduce wall-clock time.**

For a twelve-file code review, the token cost is approximately the same whether the files are reviewed sequentially or in parallel. The difference is time: twelve sequential reviews might take six minutes; twelve parallel reviews with convergence might take forty-five seconds.

8.3 When Swarms Are Worth It

Swarm execution adds overhead: worker lifecycle management, convergence processing, correlation tracking, and the convergence inference call. This overhead is justified when:

1. **N is large enough** — Below approximately four items, the coordination overhead exceeds the time savings. The exact threshold depends on the task duration and the convergence strategy.
2. **Items are truly independent** — If worker outputs depend on each other (worker 3 needs worker 1's result), the task is not suitable for swarm execution. Dependencies require sequential execution or a more complex coordination model that is outside this architecture's scope.

3. **Wall-clock time matters** — If the parent process is executing autonomously and the user is not waiting, sequential execution may be acceptable. Swarms are most valuable in interactive workflows where latency directly impacts the user experience.

✦ ✦ ✦

9. Distributed Execution

9.1 Containment Enables Distribution

The three containment rules — no joining other swarms, no initiating new swarms, no lateral communication — were introduced in Section 3 as safety constraints. They prevent the unbounded execution expansion that makes multi-agent systems unreliable. But these same constraints produce a second, more consequential property: they make workers location-independent.

Consider what a worker requires to execute:

- A task blueprint (a YAML document)
- A compiled context package (a text payload)
- An input item (a data structure)

Consider what a worker does not require:

- Access to the coordinator's memory chains
- Knowledge of other workers' existence

- A shared filesystem, database, or message bus with sibling workers
- Physical proximity to the coordinator or to other workers

The containment rules guarantee that a worker's execution boundary is closed. It reads its input, executes its task, and produces its output. It does not reach outside its boundary for anything. This means the boundary can be located anywhere — on the same machine as the coordinator, on a server across the network, on a cloud instance across the continent, or on a device on the other side of the planet.

This is not a deployment convenience. It is an architectural property that emerges from the containment model. Distribution is not something added to swarms — it is something the containment rules make structurally possible by eliminating every requirement for co-location.

9.2 Execution Topologies

A swarm's workers can be distributed across any combination of execution environments. The coordinator dispatches task instances to workers based on a placement strategy; the workers execute and return results; the convergence step collects results regardless of origin. The coordinator does not care

where a worker runs. It cares that the worker started, that the worker finished, and what the worker produced.

This produces several natural topologies:

Local swarm. All workers execute on the same machine as the coordinator. This is the simplest topology — workers are threads or processes on the local runtime. Appropriate for development, testing, and workloads where the machine has sufficient resources.

Cloud-burst swarm. The coordinator runs locally; workers execute on cloud instances. When a fan-out is large — fifty files to review, a hundred records to process — the local machine may not have the compute, memory, or API rate limits to run fifty workers simultaneously. Cloud-burst swarms dispatch workers to cloud instances that spin up for the duration of the swarm and shut down after convergence. The coordinator manages the lifecycle; the workers are ephemeral.

Edge swarm. Workers execute on edge devices or remote machines. A swarm analyzing sensor data from twelve factory floors dispatches one worker per floor, executing on local infrastructure close to the data. The compiled context package travels to the edge; the result travels back. The raw data never leaves the floor.

Federated swarm. Workers execute on machines owned by different participants. A research swarm analyzing datasets held by different institutions dispatches workers to each institution's infrastructure. Each worker sees only its local dataset through the compiled context scoping. No institution's data leaves its network. The convergence step operates on results — summaries, classifications, extracted features — not on raw data.

Hybrid swarm. Workers execute across a mixture of local, cloud, and edge environments based on input characteristics. A worker processing a small text file runs locally. A worker processing a large image dataset routes to a cloud GPU instance. A worker processing sensitive financial data routes to an on-premises secure enclave. The placement strategy makes the routing decision; the worker executes identically regardless of location.

9.3 The Compiled Context Boundary as Security Boundary

In a distributed swarm, the compiled context package is the only information that crosses a network boundary on the way in. The worker's result is the only information that crosses on the way out. This is not a coincidence — it is a direct consequence of the CCR's compilation model.

The compiled context package is precision-scoped to the current task step. It does not contain the coordinator's full memory. It does not contain other workers' inputs. It does not contain the process definition's internal metadata. It contains exactly what the worker needs to execute its task — nothing more.

This scoping produces a security property that is absent from most distributed agent systems: **the worker cannot leak what it was never given.** A worker dispatched to a remote environment to analyze a single file receives the compiled context for that file. It does not receive the contents of other files, the PR's broader context, or the organizational knowledge that informed the process definition. If the remote environment is compromised, the exposure is limited to one compiled context package and one input item.

In the federated topology, this property becomes essential. When workers execute on infrastructure controlled by different parties, each party must trust that the dispatched work does not carry unauthorized information. The compiled context boundary provides that guarantee structurally — not through access control lists, not through encryption alone, but through the architecture's fundamental design: the worker receives a minimal, scoped payload because the CCR's compilation pipeline produces minimal, scoped payloads. The security property is not bolted on. It is intrinsic.

9.4 Model Selection Across Geographies

The CCR's dynamic model selection, described in the CCR whitepaper, takes on new dimensions in distributed swarms. When workers can execute anywhere, the model selection decision becomes a joint optimization across three variables:

Capability. The task requires a specific level of reasoning depth, code generation ability, or domain knowledge. Not all models satisfy the requirement.

Locality. The input data may have residency requirements. Financial data must be processed in-jurisdiction. Healthcare data must remain within HIPAA-compliant infrastructure. A model running in the right geography may be preferable to a more capable model running in the wrong one.

Cost. Cloud GPU instances in different regions have different pricing. Local models have zero marginal inference cost but limited capability. The optimal routing minimizes total cost while satisfying capability and locality constraints.

In a distributed swarm, these three variables are evaluated per worker, not per swarm. Worker 1, processing a small text file, routes to a local model at zero marginal cost. Worker 2, processing a complex architectural analysis, routes to a cloud-hosted reasoning model. Worker 3, processing data subject to EU data residency rules, routes to a model hosted in the EU region. All three participate in the same swarm.

All three produce results that converge through the same merge step. The convergence step does not know or care which model each worker used — it operates on results, not on execution metadata.

9.5 Latency and the Geography of Work

Sequential execution has a fixed latency profile: total time equals the sum of individual step durations. Distributed swarm execution introduces a different profile: total time equals the maximum worker duration plus network round-trip time plus convergence duration.

For local swarms, network time is negligible. For cloud-burst swarms, network time is measurable but small relative to inference time — a compiled context package is kilobytes, not gigabytes. For edge and federated swarms, network time can be significant, particularly when workers are geographically distant.

The architecture handles this through deadline-aware scheduling. The process definition's `fan_out` block may specify a deadline:

```
fan_out:
  over: input_items
  task: analyze_item
  concurrency: 20
  deadline: 30s
  convergence:
    strategy: collect
    min_success_ratio: 0.8
    failure_action: skip
```

The runtime uses the deadline to make placement decisions. If a worker dispatched to a remote location is unlikely to complete within the deadline (based on historical latency data), the runtime places it closer — on a cloud instance in a nearer region, or on the local machine — even if that placement is suboptimal on other dimensions. The deadline constrains the placement strategy, ensuring that distribution does not sacrifice responsiveness beyond the process definition's tolerance.

9.6 The Implications of Location-Independent Execution

The architectural consequence of location-independent workers extends beyond performance optimization. It changes what swarms can be used for.

Global-scale analysis. A swarm can dispatch workers to every continent simultaneously. A compliance review that must evaluate operations under twelve different regulatory frameworks dispatches twelve workers, each executing in the

relevant jurisdiction, each using models trained on or fine-tuned for local regulatory language. The convergence step produces a unified compliance report from twelve jurisdiction-specific analyses, none of which required data to cross jurisdictional boundaries.

Collaborative execution without shared infrastructure. Two organizations working on a joint project can participate in the same swarm without sharing infrastructure, credentials, or raw data. Organization A runs workers on its infrastructure; Organization B runs workers on its infrastructure. The coordinator (running on either side, or on neutral infrastructure) dispatches inputs and collects results. The containment rules guarantee that neither organization's workers access the other's data or systems.

Hardware-aware routing. Some tasks benefit from specific hardware. A worker analyzing a large codebase benefits from fast local storage. A worker generating images benefits from GPU acceleration. A worker performing symbolic reasoning benefits from high-memory CPU instances. The placement strategy routes workers to hardware that matches their task profile, turning a homogeneous swarm (same task blueprint) into a hardware-heterogeneous execution with performance characteristics optimized per worker.

Resilience through geographic distribution. A swarm distributed across three cloud regions survives the failure of any single region. When workers are location-independent and the task is idempotent, the retry mechanism can resubmit failed inputs to workers in surviving regions. The swarm completes — slower, perhaps, but completely — even under partial infrastructure failure.

Progressive capability deployment. When a new model is deployed in one region but not yet available globally, distributed swarms can route specific workers to the new model while others continue using the existing model. The convergence step does not distinguish between results from different models. This enables gradual rollout of model upgrades without requiring global synchronization.

9.7 Trust and Verification in Distributed Swarms

When workers execute on infrastructure you do not control, the question of trust becomes concrete. Can you trust a worker's result? Can you verify that the worker executed the task faithfully?

Swarm Architecture addresses this through three mechanisms:

Result validation. The convergence strategy can include validation logic that checks worker results against expected schemas, value ranges, or consistency conditions. A worker that returns a result outside

expected bounds is flagged — its result can be excluded from the merge, retried on trusted infrastructure, or escalated for review.

Redundant execution. For high-stakes tasks, the same input can be dispatched to multiple workers on different infrastructure. If two workers produce consistent results, confidence is high. If they diverge, the convergence step can apply a tiebreaker (third worker, human review, or conservative default). This is the same principle as consensus in distributed systems, applied at the task level rather than the protocol level.

Execution attestation. Workers can produce signed execution records — cryptographic attestations of what input they received, what model they used, what output they produced, and what timestamp they completed. These attestations are collected during convergence and stored with the swarm's execution history. They do not prevent a compromised worker from producing a false result, but they provide an audit trail that makes falsification detectable after the fact.

These mechanisms are not required for all swarms. A local swarm on trusted infrastructure needs none of them. A federated swarm across organizational boundaries may require all three. The process

definition specifies the appropriate level of verification for each swarm, matching the trust model to the deployment topology.

✦ ✦ ✦

10. Limitations and Future Work

10.1 Deliberate Constraints

Swarm Architecture deliberately excludes several capabilities that might seem natural extensions:

No nested swarms. A swarm worker cannot initiate a sub-swarm. If a task requires nested parallelism, the process definition must express it as sequential swarm steps in the parent process. This constraint preserves containment and bounds the total parallelism to what is explicitly specified.

No inter-worker communication. Workers cannot share intermediate results, coordinate strategies, or negotiate resource allocation. If a task requires coordination between parallel workers, it is not suitable for swarm execution — it requires a different architectural pattern.

No dynamic work distribution. The input set is fixed at fan-out time. Workers cannot discover additional work items during execution. If the total

work is not known at fan-out time, the process must use a different pattern (e.g., a loop with dynamic termination conditions).

10.2 Areas for Future Investigation

Hierarchical swarms. Some tasks have natural two-level parallelism: fan out across files, and within each file fan out across functions. The current architecture handles this through sequential swarm steps, but a hierarchical model could express it more naturally while maintaining containment guarantees.

Adaptive concurrency. The current model uses a fixed `concurrency` parameter. An adaptive model could monitor worker performance and adjust concurrency dynamically — scaling up when workers complete quickly, scaling down when API rate limits are hit.

Cross-swarm learning. Currently, each swarm's execution is independent. A learning mechanism that analyzes patterns across swarms — which tasks benefit from parallelism, what concurrency levels produce the best cost/latency trade-offs, which convergence strategies produce the highest-quality results — could inform future process definitions.

Heterogeneous workers. The current model requires all workers to execute the same task blueprint. A heterogeneous model could assign

different blueprints to different workers based on input characteristics, enabling specialization within a swarm.

✦ ✦ ✦

11. Conclusion

Swarm Architecture extends the Compiled Context Runtime with a model for bounded parallel agent execution. It solves one problem precisely: naturally parallel work should execute in parallel. It does not attempt to solve general multi-agent coordination, emergent agent collaboration, or distributed consensus.

The architecture's value lies in its constraints as much as its capabilities. Containment rules prevent the unbounded execution expansion that plagues multi-agent systems. Convergence protocols make parallel result merging explicit and auditable. Correlation IDs preserve the traceability that makes the CCR trustworthy. And — perhaps most significantly — the same containment rules that make swarms safe also make them distributable. A worker that cannot reach outside its boundary can execute anywhere without risk.

This is the paper's central architectural insight. Constraints designed for safety produce a property — location independence — that transforms the scope of

what agent systems can do. A local developer parallelizing a code review and a multinational organization distributing compliance analysis across twelve jurisdictions use the same architecture, the same containment model, the same convergence protocols. The difference is topology, not mechanism.

The compiled context boundary reinforces this at the security layer. Workers receive precisely what they need and nothing more — not because of access control lists or network segmentation, but because the CCR's compilation pipeline produces minimal, scoped payloads by construction. Security is not a feature added to distribution. It is a property inherited from the context model.

One process definition. One execution model. Workers that can run anywhere — on your laptop, in your cloud, on your partner's infrastructure, on a device across the planet — because the architecture guarantees they need nothing from each other and can leak nothing they were never given.

✦ ✦ ✦

References

1. Anderson, W.C. (2026). *Compiled Context Runtime: Process-Driven Agent Execution with Unbounded Local Memory.* Version 1.0.

2. Anderson, W.C. (2026). *Volatility-Based Decomposition in Software Architecture: A Practitioner-Oriented Articulation*. Version 1.0.
3. Anderson, W.C. (2026). *Harmonic Design: A Unified Software Engineering Framework.* Version 1.0.