What Requests Taught Me About Marriage

March 2026

13 min read • 2,953 words

Themes: Consciousness Mental Health Programming Recursive Spiritual

I have spent fifteen years thinking about how humans interact with systems. It started with HTTP libraries and somehow ended up at consciousness research, but the middle ground—the place where this philosophy gets tested most honestly—is marriage.

This isn't a forced metaphor. I genuinely think this way. When Sarah and I are navigating a disagreement about whose turn it is to get up with Malachi at 3am, some part of my brain is processing the interaction in terms of API contracts and error handling. Not because I'm emotionally broken (well, not only because of that), but because the principles that make software humane turn out to be the same principles that make relationships work.

The "for humans" philosophy didn't start with HTTP and end with HTTP. It started with noticing that most systems force people to adapt to machine logic instead of the other way around. Marriage is a system. A beautiful, terrifying, irreducibly complex system that two people build together and then have to maintain for decades without any documentation.

So here's what I've learned from building the most popular HTTP library in the world, applied to the most important relationship in my life.

Sensible Defaults

When I designed Requests, the most important decision was what should happen when the user doesn't specify anything. What are the defaults?

```
import requests

# This just works.
# SSL verification: on. Redirects: followed.
# Encoding: auto-detected. Timeout: reasonable.
# You don't have to think about any of it.
response = requests.get('https://example.com')
```

The magic of good defaults is that they prevent problems before they start. You don't have to configure SSL verification because the library assumes you want security. You don't have to handle redirects manually because the library assumes you want to end up where the URL actually points. The common case just works.

> In software, good defaults encode the maintainer's wisdom about what most users need. In marriage, good defaults encode the couple's shared understanding of what keeps things running. Both require ongoing maintenance as conditions change.

Marriage needs the same thing. You need shared defaults—baseline assumptions that prevent most arguments before they start. Who handles what in the morning. What "I'll be home late" means (and at what hour it triggers a text). Whether the dishwasher gets run when it's full or when someone needs a clean fork. Whether silence means "I'm angry" or "I'm thinking."

Sarah and I had to establish these defaults the hard way, through years of failed implicit negotiations. Early in our relationship, my default for processing difficult emotions was withdrawal—go quiet, think it through internally, emerge with a conclusion. Sarah's default was to talk it through in real time. Neither was wrong. But without establishing a shared default, every emotional moment became a protocol mismatch.

```
# The early days: protocol mismatch
kenneth_default = "process internally, respond later"
sarah_default = "talk through it now"

# Result: she thinks I'm shutting her out,
# I think she's not giving me space to think.
# Both interpretations are valid.
# Neither is the full picture.
```

The fix wasn't for one of us to adopt the other's default. It was to establish a new shared default: I say "I need to think about this, I'll come back to you in an hour," and she says "okay, I'll be here." The default includes a timeout and a callback. Good API design.

Most arguments between couples aren't about the actual issue. They're about mismatched defaults—one person assuming something that the other person never agreed to. Good defaults, established consciously and revisited regularly, prevent the vast majority of domestic bugs.

Clear Contracts

Requests has a clean, explicit API. You know exactly what you're getting:

```
response = requests.get('https://api.example.com/data')

# The contract is clear:
# - response.status_code tells you what happened.
# - response.text gives you the content, decoded.
# - response.json() parses it if it's JSON.
# - response.ok tells you if things went well.
# No surprises. No hidden state. No guessing.
```

Compare this to urllib2, which required you to construct request objects, manually handle encoding, chain openers together, and generally perform a small ritual before you could make a simple HTTP call. The API was technically complete but humanly unreadable.

Implicit expectations are the urllib2 of relationships.

> I realize comparing your early communication style to urllib2 is a specific kind of insult that maybe three hundred people on Earth would fully feel. But if you're reading this essay, you're probably one of them.

"You should have known I was upset." Why? What was the status code? What was the response body? If you didn't tell me, I didn't know. I'm not a bad partner for missing an implicit signal any more than a developer is a bad programmer for not understanding urllib2's twelve-step process for making a GET request. The interface was unclear.

Sarah taught me this, actually, more than I taught her. She's better at explicit communication than I am, probably because she didn't spend her formative years talking to computers. When she needs something, she says it. Not with drama or demands—just clarity. "I need help with bedtime tonight. I'm running on empty." Clear request. Clear parameters. I know exactly what's being asked and can respond honestly about whether I can deliver.

The contract I try to honor in return: say what I mean, mean what I say. If I'm struggling—with work, with mental health, with the ordinary weight of existing—I name it explicitly rather than hoping she'll reverse-engineer my mood from behavioral clues. "Explicit is better than implicit" isn't just a line in the Zen of Python. It's marriage advice.

Graceful Error Handling

Every system breaks. The question isn't whether errors will happen—it's how the system handles them when they do.

Requests raises clean exceptions with useful information:

```
try:
    response = requests.get('https://example.com', timeout=5)
    response.raise_for_status()
except requests.ConnectionError:
    # Clear: the connection failed.
    # You know what went wrong and can respond appropriately.
    print("Could not connect. Check your network.")
except requests.Timeout:
    # Clear: it took too long.
    # Different problem, different response.
    print("Request timed out. Try again or increase timeout.")
except requests.HTTPError as e:
    # Clear: the server responded with an error.
    # The status code tells you exactly what kind.
    print(f"Server error: {e.response.status_code}")
```

The exceptions are specific, informative, and actionable. You know what went wrong and you have enough information to decide what to do about it. Compare this to a system that just crashes with a stack trace—raw bytes of panic dumped to the console, technically containing all the information but humanly useless in the moment of crisis.

Marriage errors work the same way.

"I feel hurt when you check your phone during dinner" is a well-formatted exception. It names the error (hurt), identifies the trigger (phone at dinner), and implies the fix (put it away). Your partner can catch this exception and handle it gracefully.

Screaming is a stack trace dump. All the information is technically there—the pain, the frustration, the accumulated resentments—but it's delivered in a format that's impossible to parse under pressure. The receiver's only option is to catch a generic exception and hope for the best, or crash entirely.

```
# Well-formatted relationship error
class RelationshipError(Exception):
    def __init__(self, feeling, trigger, need):
        self.feeling = feeling
        self.trigger = trigger
        self.need = need
        super().__init__(
            f"I feel {feeling} when {trigger}. "
            f"I need {need}."
        )

# vs. the stack trace dump
raise Exception(
    "EVERYTHING IS WRONG AND YOU NEVER LISTEN "
    "AND THIS IS JUST LIKE THAT TIME IN 2019 "
    "AND YOUR MOTHER DOES THE SAME THING"
)
# Good luck parsing that at 11pm.
```

I'm not always good at this. Schizoaffective disorder means my error handling sometimes degrades under stress—the well-structured exceptions get replaced by raw emotional output that even I can't parse in the moment. Sarah has learned to recognize when my error handling has degraded and to wait for the system to stabilize before trying to debug. That patience is a form of love I didn't know I needed until I had it.

Backwards Compatibility

This one matters more than people realize, in both software and marriage.

When Requests went from version 1.x to 2.x, we maintained backwards compatibility as much as humanly possible. Why? Because people had built their lives—their applications, their businesses, their workflows—around the existing API. Breaking changes aren't just inconvenient. They're a form of betrayal. You promised an interface, people trusted that promise, and now you're telling them the thing they depended on no longer works.

> Semantic versioning is, at its core, a promise about how you'll manage change. Marriage vows are similar—a semantic contract about what will remain stable even as everything else evolves. Major version bumps happen, but they should be rare and well-communicated.

Marriage is the same. Your partner has built their life around certain promises and patterns. The way you always make coffee in the morning. The fact that Sunday is family day. The understanding that you'll be honest even when it's uncomfortable. These aren't just habits—they're the API your partner depends on.

When change is necessary—and it will be, because people grow and circumstances shift—you don't just push a breaking change to production. You issue deprecation warnings first. "Hey, I've been thinking about changing careers. Nothing's decided yet. I just want you to know it's on my mind." That's a deprecation warning. It gives your partner time to adjust their expectations, ask questions, and prepare for the change.

What you don't do is come home one day and announce you've quit your job and enrolled in art school. That's pushing a breaking change to production without warning, and it doesn't matter how good the new version is—the trust violation of the unannounced change will overshadow everything.

I've learned this the hard way. There were times early on when I'd make significant decisions—about work, about spending, about how I was managing my mental health—and present them to Sarah as done deals. Technically, they were often the right decisions. But the process was wrong. I'd skipped the deprecation cycle, and the surprise did more damage than the change itself.

Human-Readable Responses

One of Requests' most beloved features is how it handles response content:

```
response = requests.get('https://api.example.com/data')

# Not this (the raw bytes, technically complete):
raw = response.content  # b'{"name": "Kenneth", "status": "tired"}'

# This (decoded, parsed, human-ready):
data = response.json()  # {'name': 'Kenneth', 'status': 'tired'}

# Or simply:
text = response.text  # Already decoded. Already readable.
```

The library does the translation work for you. It takes the raw bytes of the HTTP response and converts them into something a human can actually read and use. The information was always there in the raw response—but presenting it in a human-readable format is what makes it useful.

This is one of the hardest skills in marriage: translating your internal experience into something your partner can actually parse.

My internal experience often arrives as raw bytes—undifferentiated anxiety, unnamed frustration, a general sense of wrongness that hasn't been decoded into specific feelings yet. If I dump that raw response on Sarah, she can't do anything with it. "Something's wrong but I don't know what" is `response.content`—technically the truth, but not useful.

The work of emotional maturity is learning to be your own decoder. To take the raw bytes of internal experience and translate them into `response.text`—something readable, something your partner can work with. "I'm anxious about the meeting tomorrow and I think it's making me irritable. I'm sorry if I seem short. It's not about you." That's a decoded, human-readable response.

Sarah is extraordinarily good at reading my raw output, honestly. She's developed an intuition for my internal states that I sometimes lack for myself—the way a good HTTP client learns to handle encoding quirks that the server doesn't properly declare. But I try not to rely on that. She shouldn't have to be an expert parser of my undocumented emotional API. I should do the encoding work myself.

> The fact that she's willing to parse my raw bytes doesn't mean I should keep sending them. Love isn't an excuse for laziness in communication—it's a reason to work harder at it.

Connection Pooling

This is the one that took me longest to understand, both in HTTP and in marriage.

Requests uses connection pooling through Session objects. Instead of establishing a new TCP connection for every single request—the full handshake, the SSL negotiation, the whole expensive setup—it reuses existing connections. The context is maintained. The overhead is eliminated. Communication becomes efficient because you're not starting from scratch every time.

```
# Without sessions: every request starts from zero.
requests.get('https://api.example.com/users')    # New connection.
requests.get('https://api.example.com/posts')    # New connection.
requests.get('https://api.example.com/comments') # New connection.

# With sessions: shared context, maintained state.
session = requests.Session()
session.headers.update({'Authorization': 'Bearer trust-built-over-years'})

session.get('https://api.example.com/users')    # Reuses connection.
session.get('https://api.example.com/posts')    # Shared context.
session.get('https://api.example.com/comments') # No renegotiation.
```

Marriage is a Session object. The entire point is that you've done the handshake. You've established trust. You've negotiated the shared headers—the values, the assumptions, the inside jokes, the shorthand that only works because of years of shared context.

When Sarah says "it's a Tuesday kind of day," I know exactly what she means, and explaining it to you would take a paragraph. When I say "my brain is doing the thing," she knows which thing and what I need. That's connection pooling. That's the efficiency of maintained state.

The failure mode is when you stop maintaining the session. When you start treating every conversation as a new connection—no shared context, no accumulated trust, starting the SSL handshake from scratch each time. "Why would you think that?" becomes the question instead of "of course you'd think that, because I know you." Long-married couples who've stopped maintaining their session end up having the same argument over and over because neither side carries the context from the last time they resolved it.

The work of marriage is session maintenance. It's actively choosing to carry forward the context, the trust, the shared understanding—even when you're tired, even when you're angry, even when it would be easier to drop the connection and start fresh with someone who doesn't know your bugs.

The For Humans Philosophy, Applied to Love

The recursive loop that runs through all my work—code shapes minds, programmers shape code, therefore programmers shape collective consciousness—has an intimate corollary. The way I design systems shapes how I think about relationships, and the way I experience relationships shapes how I design systems.

Sarah doesn't read my code. She's never opened a pull request or debugged a stack trace. But she's the best interface I've ever worked with—not because she's simple (she's the furthest thing from simple), but because she's honest. Her API is clear, well-documented through years of explicit communication, and the error messages always tell me what I need to know even when I don't want to hear it.

I think about programming as spiritual practice and I realize that marriage might be the most demanding spiritual practice there is. It requires every skill I've developed as a programmer—patience with complex systems, humility when my model of reality is wrong, the discipline to write clean code even when I'm tired—and applies them to the highest-stakes system I'll ever maintain.

Malachi is watching how we interact. He's three, and he's already learning defaults from us. How people handle conflict. What it looks like when someone says "I'm sorry" and means it. Whether love is performed or practiced. We're not just maintaining our own system—we're writing the initial codebase that our son will fork and modify for the rest of his life.

That's the recursive loop at its most personal. The values I embed in how I treat Sarah become the values Malachi absorbs as normal. The defaults we model become his defaults. The error handling patterns he witnesses become his first template for navigating relationships.

No pressure.

What I'm Still Learning

I want to be honest about something: I'm better at designing humane APIs than I am at being a humane partner. It's easier to write clean interfaces for HTTP than for the person sleeping next to you, because HTTP doesn't have feelings about your response time and the protocol doesn't remember that you forgot its birthday in 2021.

The "for humans" philosophy is aspirational in marriage the same way it's aspirational in software. You ship the best version you can, knowing it has bugs. You commit to iterating. You try not to introduce regressions. You read the error logs—the sighs, the silences, the "it's fine" that clearly isn't fine—and you debug with patience instead of defensiveness.

Some days I'm a clean, well-documented API with sensible defaults and graceful error handling. Other days I'm urllib2—technically functional but requiring so much overhead to interact with that everyone around me is exhausted by the effort. Sarah stays with me through both versions. That's not just backwards compatibility. That's grace.

> Grace: the quality of treating someone better than their current API version deserves, because you remember the version you fell in love with and you trust the version that's coming next.

The work continues. The system is never finished. The best I can do is keep the session alive, keep the defaults honest, keep translating my raw bytes into something she can read, and keep honoring the contract we've built together—not because it's technically required, but because she's the human on the other side of every interface I'll ever design.

And she always has been.

This essay explores API design principles as relationship wisdom, connecting the "for humans" philosophy that began with HTTP libraries to the most intimate system we ever build with another person. It extends themes from Programming as Spiritual Practice, The Recursive Loop, and the family life that grounds all of this work in something real. For the broader trajectory from technical design to consciousness research, see From HTTP to Consciousness.