

overload 195

OCTOBER 2026 £4.50

Type Algebra and Functional Composition in libfn

Bronek Kozicki introduces
a C++ library that
chains results and errors.

Configuration is the Deployment Nobody Canaries

Santhosh Kumar Somarapu presents a four-rung
safety model of outcome-based canaries

Load-Resilient C++ Servers

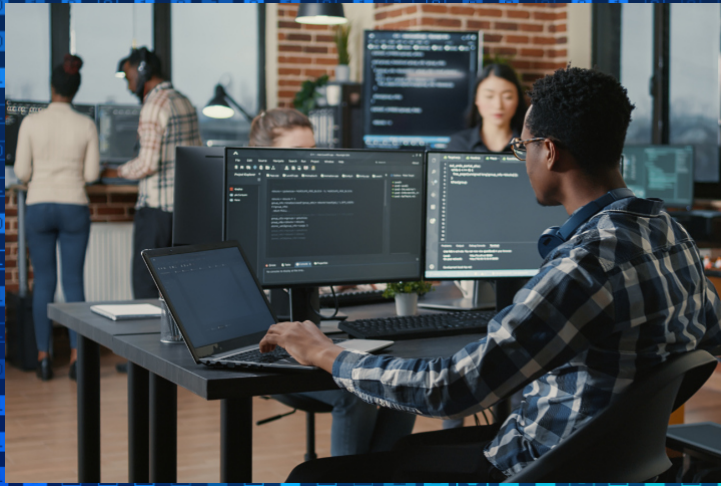
Ajay Pandey uses `select()`, bounded queues and
modern C++ concurrency to solve the problems.

Afterwood

Chris Oldwood discusses where the line is drawn
between testing thoroughly and being obsessed.

accu

Professionalism in Programming



**Professional development
World-class conference**

**Printed journals
Email discussion groups**



**Individual membership
Corporate membership**



**Visit accu.org
for details**



October 2026

ISSN 1354-3172

EditorFrances Buontempo
overload@accu.org**Advisors**

Paul Bennett
t21@angellane.org

Matthew Dodkins
matthew.dodkins@gmail.com

Paul Floyd
pjfloyd@wanadoo.fr

Jason Hearne-McGuinness
coder@hussar.me.uk

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Christian Meyenburg
contact@meyenburg.dev

Barry Nichols
barrydavidnichols@gmail.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.co.uk

Balog Pal
pasa@lib.hu

Honey Sukesan
honey_speaks_cpp@yahoo.com

Jonathan Wakely
accu@kayari.org

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover designOriginal design by Pete Goodliffe
pete@goodliffe.netCover photo by Alison Peck, taken
in the adventure playground at
Belton House, Lincolnshire, UK.**ACCU**

ACCU is an organisation of
programmers who care about
professionalism in programming. We
care about writing good code, and
about writing it in a good way. We are
dedicated to raising the standard of
programming.

Many of the articles in this magazine
have been written by ACCU members –
by programmers, for programmers – and
all have been contributed free of charge.

Overload is a publication of the ACCU
For details of the ACCU, our publications
and activities, visit the ACCU website:
www.accu.org

4 Configuration is the Deployment Nobody Canaries

Santhosh Kumar Somarapu presents a four-rung
safety model of outcome-based canaries.

8 Type Algebra and Functional Composition in libfn

Bronek Kozicki introduces a C++ library that
chains results and errors.

20 Load-Resilient C++ Servers

Ajay Pandey uses `select()`, bounded queues, and
modern C++ concurrency to solve the problems.

28 Afterwood

Chris Oldwood discusses where the line is drawn
between testing thoroughly and being obsessed.

Copy deadlines

All articles intended for publication in *Overload* 196 should be submitted by
1st November 2026 and those for *Overload* 197 by 1st January 2027.

Copyrights and trademarks

Some articles and other contributions use terms that are either registered trade marks
or claimed as such. The use of such terms is not intended to support nor disparage any
trade mark claim. On request, we will withdraw all references to a specific trade mark
and its owner.

By default, the copyright of all material published by ACCU is the exclusive property
of the author. By submitting material to ACCU for publication, an author is, by default,
assumed to have granted ACCU the right to publish and republish that material in any
medium as they see fit. An author of an article or column (not a letter or a review of
software or a book) may explicitly offer single (first serial) publication rights and thereby
retain all other rights.

Except for licences granted to 1) corporate members to copy solely for internal
distribution 2) members to copy source code for use on their own computers, no material
can be copied from *Overload* without written permission from the copyright holder.

//No comment

Comments come in many forms. Frances Buontempo considers when they are helpful and when they can be harmful.

“It’s been a contrasting couple of months. Initially, I spent far too much time lounging on the sofa watching any old nonsense on telly, including some police dramas. How many times did someone arrested say, “No comment”? Several times. Having done very little, I then volunteered for Safer Spaces [SaferSpaces] at the Leeds Festival. They are a charity trying to prevent domestic violence. They offered a space with bean bags and comfy sofas at the festival, so people could chill out and have a chat if needed. I’ve not done any youth work for a while. There were some older people, but this festival lets 16+-year-olds in, so many younger people came along. For many, it was the first time in a tent, or possibly even the first time away from home. The chill out space we provided was useful, and the staff were on hand to help. It was important to be prepared to listen rather than talk. No comment, but in a different sense to fictional characters being arrested. Trying to get back into writing after a break is a challenge, so you guessed it. I haven’t got an editorial for you, as ever.

Now, back to technical matters. I have seen differing opinions over comments in code in my career. Some people insist on writing comments, but many would rather the code spoke for itself. Sometimes inexperienced people prefer comments: it can take a while to get used to reading source code, so a human sentence or two can seem easier to read. However, the comment may not match the code because comments can be left behind as code moves on. Comments can get out of sync with code, which is surely even more confusing than no comments at all. Delete the comment, and fix the code, right? You know it makes sense. I don’t trust comments. However, I do sometimes write comments, for example giving a reference to an academic paper, if I have encoded an algorithm or calculation, so people can go read more. Comments can be useful, but there are other ways to figure out what code does, for example by writing a few tests, or doing a scratch refactor.

Nowadays, comments are often used to get ‘AI’ to write code. The comment acts as a prompt, and the code spews forth. Now, should you save these prompts? Opinions seem to be divided; for example, I have seen various discussions on Reddit [Reddit]. Trying to pick your way through a code base littered with AI prompt comments sounds like my worst nightmare. I have also heard tales of people getting AI to write documentation or README files, perhaps even comments. Now, I’m not fully qualified to have opinions on how to manage the prompts, but traditionally, we have kept the code we wrote in version control, academia notwithstanding. (Aside: I did persuade an academic to use version control once, which I count as a win.) We don’t add notes about our thought process to version control too. We may keep architectural decisions records, explaining choices and

context [Fowler26] or similar, but I’ve never seen anyone write out what they were thinking when they wrote code. Sometimes you can follow a small trail of clues to figure out how code got where it is. I found a typo in a comment once, which helped me trace which part of the internet it was copied from though. When I copied the tests over too, I found the source of a problem that had been puzzling me. Comments can have their uses!

I mentioned Reddit earlier. I have a long streak going. I have forgotten why I even started looking to be honest, as often happens with habits. Sometimes I scroll a long way down a discussion, reading the ‘the bottom of the internet’ comments. I’m sure you know what I mean: things go off on tangents, and people start being rude. It’s not all bad, but you need to pick your way through discussions if you want to join in. “Don’t feed the trolls,” as the saying goes. Now, the internet hasn’t caused trolls. Some claim people are more likely to leave anonymous comments online than actually say something rude directly to people. That may be true, but some people have always felt the need to say something they consider funny even if others find it rude. The Safer Spaces tent at the festival had hair dryers and similar on offer for free. Several young lads wandering passed asked “Where’s the Man Cave?” Oh my, aren’t you funny! When the rain started, one or two guys asked if they could use a hair dryer. Enough said.

Maybe you don’t do volunteer work. Don’t forget you can get involved with ACCU: write a book review or try out a ‘homework question’ or ‘code critique’ in the members’ *CVu* magazine. There are a few volunteer posts awaiting someone too: advertising, local groups, publicity, social-med, study-groups-officer. If you’d rather sit back and just read, that’s fine. Each to their own.

One thing I learnt of the festival was how valuable listening is. That’s an important skill at work too, particularly if you end up in a more senior role. Rather than grabbing the keyboard and driving, it’s useful to let a less experienced person take the lead. You learn when you listen, but you need to learn to listen. It’s too easy to assume you know what someone is going to say or do next, or concentrate on what you want to say rather than hear what is really being said. If you want to know more, look up ‘active listening’. I’m no expert, but you need to actually listen, avoid being judgemental, clarify, maybe by asking questions, and summarise, by restating what you think the other person said. Words are not precise, so misunderstandings can creep in. Repeating back what’s been said in your own words gives an opportunity to clear up miscommunication. It’s too easy to pass comment or give unsolicited advice.

Now, sometimes people freeze up, so you might need to say something to keep the conversation going. Just a small prompt can work. At the festival, asking which band the person had enjoyed or was looking forward to helped cheer people up or at least distract them from feeling



Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD using AI and data mining. She’s written a book about machine learning called *Genetic Algorithms and Machine Learning for Programmers*, and one to help you catch up with C++ called *Learn C++ by Example*. She has been a programmer since the 90s, and learnt to program by reading the manual for her Dad’s BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

down. Asking how someone intended to get home was useful too. You could find out if they were sorted or needed help. Perhaps you have a few handy prompts to help you code with someone. “How are we going to test this?” “What is this class for?” And so on. If you find yourself in a more senior role and a less experienced person asks a question, never be afraid to say “I don’t know. Let’s find out.” That’s much more useful than making things up! Yes, I’m looking at you, AI.

We all freeze up from time to time. Open-ended questions at interviews being top of my list for stalling mid-sentence, not that I’ve been interviewed for a long while now. Once I was asked to sketch the architecture of the system I was working on at the time. Simple, right? Well, yes. But I wanted to caveat the diagram with the comment, “Bear in mind, I am looking for another job. For exactly these reasons.” Once I’d finished drawing and explaining as I went, they told me that was a terrible design. I know, right?! With hindsight, I could have started by saying “There’s a simpler system trying to escape from the weeds.” Then I could have drawn a fictitious setup. I can see how talking about what you are working on at the moment is a good interview question. It opens up the conversation and ideally should allow discussion around best parts and what you’d like to improve. I didn’t get that job: part of the feedback was I lacked architectural skills. Fume! I didn’t get my choice in how the system worked, so was looking for a new job. Oh well. I’m sure there’s a lesson in there. At a later date, I was asked a similar question, and said I would rather talk about a system in a previous job, because the architecture was much better. That worked out OK. Asking about someone’s experience at an interview is a sensible, open-ended question. However, be prepared for them to have different experience to you. If you’re not careful, you might get the famous *Blade Runner* (1982 original) ‘Tears in the rain’ quote [IMDb]:

I’ve seen things you people wouldn’t believe. Attack ships on fire off the shoulder of Orion. I watched C-beams glitter in the dark near the Tannhäuser Gate. All those moments will be lost in time, like tears in rain. Time to die.

I personally have seen code some people might not believe. I’ll tell you more if you buy me a beer. Of course, some of the bad code was mine. You live and you learn.

I’ve seen some great code too. To be fair, I’m not sure exactly how to qualify what I mean by good code. There can be a fine line between a neat trick and far too cute. We have ideas about clean code, and try out words like SOLID or FLUID. Chris Oldwood wrote about this in his ‘KISSing SOLID goodbye’ article many years ago. [Oldwood14]. He managed to pare the five principles down to two: program to an interface and separation of concerns. Neat, though not the cutesy acronym to help you remember the principles. I’ll offer my own here: NEAT.

- Necessary
- Expressive
- Adaptable
- Testable

Comments welcome ☺

There are various types of comments. Passing comments, throw away comments, single line, block comments. The human language versions are

often more like remarks or asides, so in a sense distractions from the main point. Code comments often work like that too. People also sometimes say they aren’t qualified to comment. I’m not sure what qualifies you: I guess the context. When you only know a little about something, you might be more likely to speak up than when you know lots. The more you know, the more you might spot subtleties and nuances, realizing there’s no one neat response to make. I’ve been reading Klaus Iglberger’s *C++ Software Design* book recently [Iglberger22]. It’s a great book. He made one small remark or comment in the context of performance: “Do not rely on gut feelings.” Sometimes gut feelings are helpful, but you often need to go further and actually check the details. Even fact checking can go wrong, if you approach a topic with preconceived ideas. It’s easy and tempting to cherry pick research. This is known as ‘motivated reasoning’ [Wikipedia-1]. You can often find some data to back up your point of view, but there may be other oppositional research. An unbiased viewpoint would admit to both sides of the argument. One of my favourite comedians, Stewart Lee, once said, “You can prove anything with facts.”

Let’s avoid throwaway comments, learning when to speak and when to simply listen. In his *Tractatus Logico-Philosophicus*, Wittgenstein said, “Whereof one cannot speak, thereof one must be silent.” [Wikipedia-2] He investigated the relationship between language and reality. It’s relatively hard to read, being a list of numbered thoughts, without reasons or arguments. You could suggest the book is a sequence of passing comments. However, if you can’t figure out what to say or how to say it, maybe pause for a bit and listen. Or start an email discussion on accu-general, or write an article and get some feedback. Getting your thoughts in a straight line is hard, and writing an editorial is even harder, as I have proved many times over.

References

- [Fowler26] Martin Fowler ‘Architecture Decision Record’, posted 24 March 2026, available at <https://martinfowler.com/bliki/ArchitectureDecisionRecord.html>
- [Iglberger22] Klaus Iglberger (2022) *C++ Software Design: Design Principles and Patterns for High-Quality Software*, O’Reilly.
- [IMDb] *Blade Runner* quote: <https://www.imdb.com/title/tt0083658/quotes/?item=qt0378266>
- [Oldwood14] Chris Oldwood, ‘KISSing SOLID Goodbye’, in *Overload* 22(122):14-17, August 2014, available at https://accu.org/journals/overload/22/122/oldwood_1957/
- [Reddit] ‘How are you managing prompts in your codebase as your AI features get bigger?’, available at https://www.reddit.com/r/webdev/comments/1pgwkcc/how_are_you_managing_prompts_in_your_codebase_as/
- [SaferSpaces] ‘Preventing domestic abuse and sexual violence, against women and girls, at festivals and events through education, training and immediate specialist support’: <https://www.saferspaces.co.uk/>
- [Wikipedia-1] ‘Motivated reasoning’: https://en.wikipedia.org/wiki/Motivated_reasoning
- [Wikipedia-2] Wittgenstein: https://en.wikipedia.org/wiki/Tractatus_Logico-Philosophicus



If you read something in *Overload* that you particularly enjoyed, you disagreed with or that has just made you think, why not put pen to paper (or finger to keyboard) and tell us about it? We’d love to hear from you!

Configuration is the Deployment Nobody Canaries

Runtime configuration can change production behaviour faster than code. Santhosh Kumar Somarapu presents a four-rung safety model of outcome-based canaries.

Deploying code carefully is a solved problem, in the sense that the solutions are well known and widely available. Release a new binary to a fraction of instances, watch it, promote or roll back. The tooling is mature enough that arguments now concern which metrics should gate promotion, not whether promotion should be gated.

Now consider how a configuration change reaches production. A value changes in a control plane. It propagates in seconds. Nothing watches a fraction of traffic; nothing compares before and after; nothing rolls back automatically. The machinery we built for deployments does not apply, because a configuration change is not a deployment. There is no artefact, no image, often no restart.

The allocation of safety is therefore inverted. In most systems I have worked on, configuration changes outnumber code deployments by a large factor, take effect faster, and are made by more people, often under time pressure during an incident. The change that gets the least scrutiny is the one that happens most often and propagates fastest.

The industry has the incident record to show what that costs. Meta's outage of 4 October 2021 began with a routine maintenance command that, in Meta's own account, "unintentionally took down all the connections in our backbone network", taking the company's platforms and much of its internal tooling with it [Facebook21]. Cloudflare's outage of 2 July 2019 was a single new rule deployed to its managed WAF, whose regular expression caused catastrophic backtracking and "CPUs to become exhausted on every CPU core that handles HTTP/HTTPS traffic" across the network, for 27 minutes [Cloudflare19]. Google Cloud's networking incident of 2 June 2019 degraded multiple regions when existing configuration, network control-plane jobs marked as eligible to be stopped during a rare class of maintenance event, met a bug in the software initiating those events that descheduled clusters in several physical locations at once [Google19]. None were bad binaries. All were changes that were well formed, permitted, and applied exactly as instructed.

A word on scope. This article concerns dynamic runtime configuration: independently mutable values distributed through a control plane while the service keeps running. Such changes often lack the separately promotable, immutable artefact that code-deployment pipelines rely on, and can propagate rapidly across a large population. Configuration that ships inside a build, and is promoted with it, already inherits the deployment pipeline's protections and is not the subject here.

What follows is the Configuration Safety Ladder, four rungs I have found useful for placing a team's configuration safety and for arguing about what to build next. Its purpose is to separate two questions that usually

get conflated: whether a change is correct, which the industry is good at, and whether its effect is plausible, which almost nothing checks. The interesting jump on the ladder is not the one most teams expect. Getting from 'no safety' to 'some safety' is comparatively easy. Getting from 'some safety' to 'safety that catches the failures that actually matter' is where the difficulty lies, and that difficulty turns out to be statistical rather than architectural.

The article makes three claims. Validity and effect are different questions, and a configuration change can be entirely valid while its effect is wrong, which is why correctness checking does not catch this class of failure. Outcome gating is usually limited by statistical information rather than by integration effort, so the constraint is the sample the window can supply. And autonomous reversal is safe only when both the prior state and the reverse operation are well defined, which for configuration is frequently not the case.

Why configuration is structurally different

Three properties distinguish a configuration change from a code change, and each removes a defence we normally rely on.

There is often no independently promotable artefact. A binary can be built, scanned, signed, staged and promoted through environments, because it is a thing that exists and can be pointed at. A runtime configuration change is frequently a value in a database row or a key in a distributed store. The pipeline has nothing of its own to carry.

It can take effect rapidly and broadly. Deployment tooling is built around the assumption that rollout takes time, and that time is where the safety lives, because you get to look before the change is everywhere. Configuration propagation is designed to be fast and uniform, which is exactly right when the change is correct and exactly wrong when it is not.

The failure is often not a crash. This is the important one. A bad binary tends to announce itself: it panics, fails a health check, the process restarts, the supervisor notices. A bad configuration frequently produces a perfectly healthy system doing the wrong thing. Every component is up, every request returns 200, and the behaviour is wrong.

That third property is what makes the rest of this article necessary. If configuration failures looked like crashes, existing health checks would catch them and there would be nothing more to say.

The configuration safety ladder

The ladder is cumulative shorthand rather than a claim that detection quality and recovery automation are the same dimension. They are partially independent, and a system can have autonomous health-based roll-back without any outcome gating. Each rung here assumes the foundations of the ones before it.

Rung 1 – Manual. A human reviews the change and applies it. In many systems I have encountered, most configuration lives here. Review catches typos and obvious mistakes; it does not catch a value that is syntactically

Santhosh Kumar Somarapu is a software engineer working on large-scale distributed systems and infrastructure reliability, with experience at Meta, Amazon, Microsoft, Oracle and eBay. Over the course of his career, he has built configuration-safety and change-management systems for high-traffic data platforms. He holds an MS in Computer Science from the University at Buffalo. Contact: somarapu.santhosh91@gmail.com

Health signals answer ‘is the system operating’. Configuration failures are frequently ‘the system is operating on the wrong instructions’.

fine and semantically wrong, and it does not catch interactions with state that has drifted since the value was last considered.

Rung 2 – Health-gated. Apply the change to a subset, watch system health (error rates, latency, saturation, process liveness) and revert if something moves. This is a genuine improvement, and teams that invest in configuration safety often stop here. It catches the loud failures: a malformed value that makes a service throw, a limit set so low that requests are rejected.

Rung 3 – Outcome-gated. Watch what the system is *achieving* rather than whether it is running. For a commerce system: are orders being placed. For an advertising system: are ads being served and recorded. For a data pipeline: is the volume arriving downstream what we expect. Promote on the business signal, not on the absence of errors.

Rung 4 – State-coherent autonomous recovery. The gate reverts without a human, and the blast radius is bounded by construction rather than by how quickly someone notices. It is tempting to describe this as rung 3 plus the courage to let it act. That is how I used to describe it, and it is wrong in a way that matters, for reasons I will come back to once the harder problem is out of the way.

The jump that matters

The gap between rungs 2 and 3 is where the failures live, and it is worth being concrete about why health checks do not close it.

Consider a configuration value that controls which of several downstream providers a request is routed to, and a change that misroutes a category of traffic to a provider that accepts it and returns success for everything. Error rate: unchanged. Latency: possibly improved, since the wrong provider may be doing less work. Saturation: unchanged. Every health signal is nominal. The system is in perfect health and is not doing its job.

Or a sampling rate that quietly drops most of the telemetry a downstream system depends on: the pipeline reports success, because dropping data is what it was told to do.

Health signals answer ‘is the system operating’. Configuration failures are frequently ‘the system is operating on the wrong instructions’. No amount of refinement to the first will answer the second, so you cannot reach rung 3 by improving health checks. The failure is invisible in the class of signal they measure.

Why outcome-gating is hard: the window

The obvious objection to outcome-gating is architectural. How do you get business metrics into a configuration pipeline? In practice that is an integration problem, and integration problems yield to effort.

Once the integration exists, statistical information usually becomes the binding constraint. Engineering can improve efficiency, but it cannot remove the underlying sample requirement.

To gate a change on an outcome, you must decide, within a canary window, whether the observed outcome for the canary population differs from the

control population by more than chance. Failure to detect a difference is not evidence that the change is safe: an underpowered test simply fails to see harm that is there. Promotion should require enough evidence to rule out degradation beyond a declared tolerance, and an inconclusive result should pause or escalate the rollout rather than promote it. That requires enough events in the window to distinguish a real effect from noise.

The calculation below is a detectability approximation for a two-sided difference test, not the promotion rule itself. A production gate meant to rule out unacceptable degradation should use a one-sided non-inferiority criterion against a declared harm margin; the sample-size scaling is similar, and it is the scaling that carries the argument here. And the window cannot be long, because a canary that runs for hours weakens the blast-radius advantage it exists to provide, and becomes operationally close to a prolonged partial rollout with the bad configuration live throughout.

So there is a direct tension: **statistical confidence wants a long window; blast-radius limitation wants a short one.** Everything difficult about rung 3 follows from that.

Some arithmetic makes the shape clear. For a rate-like metric, the sample size needed per arm is approximately

$$n \approx \frac{16p(1-p)}{(p\delta)^2}$$

where p is the baseline proportion of the metric being watched, δ is the smallest relative change we want to be able to detect, and n is the number of observations required in each arm, canary and control. The constant 16 comes from the standard normal approximation at 80% power and a 5% two-sided significance level [Cohen88].

This is a rule of thumb rather than a precise result. It assumes independent observations and a two-sided test, and real traffic violates independence in interesting ways, correlated as it is by user, by session and by region. The scaling is what matters here, not the constant.

Take a checkout conversion rate of $p = 0.02$, and suppose we want to catch a 10% relative drop, so $\delta = 0.1$. Then

$$n \approx \frac{16 \times 0.02 \times 0.98}{(0.002)^2} \approx 78,400 \text{ observations per arm}$$

If the canary arm receives 1% of traffic, and the service handles 500 requests per second, the canary sees 5 per second, and 78,400 observations takes over four hours.

Four hours is not a canary window. And note what happens if we try to fix this by turning the knobs:

- **Widen the canary to 10% of traffic** and the window falls to about 26 minutes, but the blast radius is now ten times larger, which is the thing we were trying to limit.
- **Accept a coarser detection threshold**, catching a 30% drop instead of 10%, and the required sample falls by roughly a factor of nine, to about 8,700 per arm. At a 1% canary allocation the window

falls from over four hours to roughly 29 minutes; only at a 10% allocation does it reach about three minutes. But we have chosen not to catch a 10% revenue drop, which for many businesses is a very large number.

- **Pick a higher-frequency metric.** Requests served per second gives orders of magnitude more observations than completed checkouts. But it is further from the outcome we care about, and the failures we are hunting are precisely the ones that leave the high-frequency signals intact.

There is no configuration of those three that is good on all axes. One reason teams stop at rung 2, in my experience, is not a shortage of engineering effort: they build the outcome-gating machinery, discover the windows required are impractical for their traffic volume, and quietly fall back to health checks.

Living with the constraint

Several things help, none of which removes the tension.

Choose metrics by variance, not by importance. The metric that best expresses business value is often the one with the least favourable statistics: rare, high-variance, slow. A proxy that sits on the same causal path and is an order of magnitude more frequent may detect the same failure far sooner. Completed orders are what the business cares about, but payment attempts and checkout-step progression lie downstream of the same routing change, move with it, and arrive with far more observations per minute. The right question is not “what matters most” but “what moves with the failure and moves measurably”.

Restrict deliberately, then randomise. Assigning uniformly across the whole population maximises generality and minimises detection speed, because it spreads the canary thinly across traffic the change cannot affect. Instead, select the region, traffic class or customer population in which the configuration is expected to have an effect, then randomly assign eligible units between canary and control *within* that population. Randomisation inside the restricted population is what keeps the comparison valid; the restriction is what concentrates the effect and shortens the window. It costs generality, since a canary on one segment tells you about that segment, but configuration changes frequently affect a specific slice of behaviour, so that is usually the right trade.

Use sequential tests, not fixed-window tests. A fixed- n test wastes the case where the change is catastrophic. If the canary’s outcome collapses you know within seconds, so there is no reason to wait for the sample size a subtle change would need. Sequential methods evaluate continuously with a boundary that controls false positives under repeated looks [Wald47], [Johari22], giving an early exit for large effects while retaining power for small ones. They can substantially reduce detection time for large effects, but they require correctly calibrated boundaries, reliable outcome maturation and a small, predeclared set of decision metrics.

Be explicit about what you cannot detect. If your traffic volume supports catching a 30% drop in ten minutes and nothing finer, that is a fact worth writing down next to the gate. The failure mode otherwise is a team that believes it has outcome-gating and has, in effect, a very slow health check.

Quietness is a correctness property

One further constraint deserves emphasis, because it is the one most often discovered too late.

A gate that blocks legitimate changes will be disabled by the people it protects, not through recklessness but through arithmetic. If a guardrail fires on ten legitimate changes for every real failure, and each false positive costs an engineer twenty minutes at an inconvenient moment, someone will eventually route around it and they will be behaving rationally.

That makes false-positive rate a correctness requirement rather than a quality-of-life concern. A gate that is disabled protects nothing. Accuracy is the wrong number, because it flatters any detector when real failures are rare. Two quantities matter instead. The false-block rate is the proportion

of safe changes the gate blocks. Alert precision is the proportion of gate interventions that correspond to genuine problems. Operator burden can additionally be expressed as false blocks per genuine problem caught. A gate with a low false-block rate that is trusted beats a more discriminating one that has been switched off.

There is a design consequence. Much of what looks like a large change is intentional, and the system cannot tell intentional from accidental by inspecting the change. But the author can. Separating the two questions –

- *is this change large?* – computable from the diff, before anything is applied
- *was a large change expected?* – not computable, declarable only by the author, out of band

– and firing only when the answers disagree turns a noisy gate into a quiet one. The declaration can be as simple as a marker in the commit message or an annotation on the change. It costs the author one line when they mean it, and it catches the case where nobody meant it.

Why rung 4 is a different problem

I said earlier that autonomous reversion is not simply rung 3 with a human removed. The reason is that reverting a configuration change is a much weaker operation than reverting a deployment, and almost every treatment of this subject assumes otherwise.

Rolling back a deployment has a well-defined meaning. The previous artefact is immutable, it still exists, it was tested as a unit, and re-applying it returns the system to a state that demonstrably worked. None of those four properties reliably holds for configuration.

The previous value may no longer be a coherent destination. Configuration is usually a shared namespace, changed by many people at a high rate. If key A was changed an hour ago and key B ten minutes ago, reverting B alone produces a combination that has never existed and was never tested. The system does not return to a known-good state; it arrives at a new one, chosen automatically, under incident conditions. When those keys interact, say a routing weight and the pool size it feeds, the reverted state can be worse than either the old or the new one.

Reversion is itself a configuration change, so it travels the same path that just failed. If the change propagated to eighty per cent of the fleet and stalled, the revert inherits that. Worse, if the pipeline is gated on the very signals the bad change has disturbed, the revert can be blocked by the gate that noticed the problem. Any design here needs an explicit answer to whether the revert path is subject to its own guardrails, and the answer is genuinely not obvious: exempting it removes the protection precisely when the system is least healthy, and not exempting it risks deadlock.

The resolution I have settled on is to exempt the revert path but make the exemption narrow and expensive. A break-glass path should apply exactly one thing, the last known-good version, rather than arbitrary configuration. It should page on use rather than merely log, because a break-glass action nobody notices is indistinguishable from an unguarded pipeline. And it should be exercised on a schedule, as disaster recovery is: an escape hatch never opened is an assumption, not a mechanism. The narrowness is what makes it safe, since the gate is bypassed only for a state that was already serving traffic successfully.

An automated revert also needs a precondition: that the harmful revision is still the current one. If a newer revision has landed in between, reverting to the last known-good version would silently discard unrelated work, so the recovery system should pause and escalate instead. And reversion cannot undo irreversible side effects. Messages already emitted, customer communications already sent and external actions already completed require compensation rather than rollback, and working out which of your configuration surfaces have that property is part of the same design job.

Reverting can be the more dangerous direction. Consider a rate limit raised in response to legitimate growth, where the increase happens to coincide with a latency rise caused by something else. An automatic revert lowers the limit during a traffic peak and sheds load that the system

could have served. The gate observed a correlation and acted on it as though it were a cause. A human would likely have hesitated. This is the standard argument against automation, but it lands harder here, because configuration changes are frequent enough that coincidental correlation is common rather than rare.

None of this argues against rung 4. It argues that the work of rung 4 is not the gate, it is making reversion well-defined: versioning the configuration set as a whole rather than key by key, so there is a coherent artefact to return to; recording which version each consumer actually holds, so partial application is visible; and deciding in advance which changes are safe to reverse automatically and which must escalate. That is unglamorous state management, and it is the real content of the rung.

Two practices make this tractable without building much. Keeping the configuration set in version control, and having the control plane apply revisions from it rather than accepting writes directly, gives you an immutable candidate revision and an unambiguous history. It does not by itself make every combination of values semantically coherent. Coherent recovery still needs related values grouped together, cross-value invariants validated, and the revision applied atomically, or with partial application made explicitly observable. Second, validate intent rather than syntax at admission time, so the pipeline holds a declared expectation, say that a change affects one region and should not alter the number of routing destinations, against which the computed effect can be checked. That turns ‘is this well formed’ into ‘is this what its author meant’, which is the same separation the guardrail relies on.

It is also why I think rung 4 should be attempted narrowly. The changes worth reverting automatically are the ones where the reverse operation is unambiguous and cheap: a single feature gate, a single routing weight, a value with no interactions. That set is smaller than it first appears, and picking it deliberately is more useful than a general capability nobody trusts enough to leave enabled.

Where to start

If a team has nothing and wants something, the order I would suggest is not the ladder’s order.

Start by making configuration changes *observable* as changes: recorded, attributable, and visible on the same dashboards as deployments. Much of rung 2’s value comes from being able to correlate ‘the graph moved’ with ‘something changed’, and that requires no gating at all.

Then add magnitude checks before health checks. Refusing a change that would remove most of a configuration set is cheap, deterministic, needs no statistics, and catches the catastrophic class outright: the empty file, the wrong path, the accidental wildcard.

Then health-gate changes whose expected risk justifies progressive exposure, judged on blast radius, business criticality, detectability and reversibility rather than on frequency alone.

Then, and only for the configuration surfaces that carry the business outcome, do the work of outcome-gating, accepting that it will be a small number of surfaces, because the statistics only work where the traffic volume is.

Conclusion

The asymmetry is what I find hardest to defend. We built elaborate machinery for the change that happens weekly, is reviewed, and announces its failures loudly. We built almost nothing for the change that happens hourly, is reviewed lightly, propagates in seconds, and fails silently.

Closing that gap fully is genuinely hard, and I have tried to be honest about why: the statistics of short windows are unforgiving, and no amount of engineering removes the tension between confidence and blast radius. The top of the ladder turns out to be constrained by something more mundane still, which is that we rarely give ourselves a coherent state to return to.

The Configuration Safety Ladder is still worth having in mind. The first two rungs generally require less statistical infrastructure than the third, many organisations have not climbed them, and knowing which rung you are on beats believing your health checks are catching things they cannot see.

If a reader takes one thing from this, I would rather it be the diagnostic than the ladder. When a configuration change goes wrong, ask whether any signal in the pipeline could have distinguished the system working from the system working correctly. For most pipelines the answer is no.

That answer should be less acceptable than it currently is. We treat validity and permission as sufficient for the most frequent change in production, and we would not accept the same standard for a binary. Until a configuration change has to demonstrate that its effect is plausible, and not merely that it parses and is allowed, the outages above will keep happening in the same shape, and we will keep describing them afterwards as though nobody could have seen them coming. ■

References

- [Cloudflare19] Cloudflare, ‘Details of the Cloudflare outage on July 2, 2019’, 12 July 2019. <https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/>
- [Cohen88] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*, 2nd edition, Lawrence Erlbaum Associates, 1988. The approximation used here is the standard two-proportion power calculation; the constant 16 follows from 80% power at a 5% two-sided significance level.
- [Facebook21] Facebook Engineering, ‘More details about the October 4 outage’, 5 October 2021. <https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/>
- [Google19] Google Cloud, ‘Google Cloud Networking Incident #19009’, 2 June 2019. <https://status.cloud.google.com/incident/cloud-networking/19009>
- [Johari22] R. Johari, P. Koomen, L. Pekelis and D. Walsh, ‘Always Valid Inference: Continuous Monitoring of A/B Tests’, *Operations Research*, vol. 70, no. 3, 2022, pp. 1806-1821. DOI 10.1287/opre.2021.2135
- [Wald47] A. Wald, *Sequential Analysis*, John Wiley and Sons, 1947.

Advertise in CVu & Overload

80% of readers make purchasing decisions or recommend products for their organisations.

Reasonable rates. Flexible options. Discounts available to corporate members.

Contact ads@accu.org for information.

Type Algebra and Functional Composition in libfn

Large application-wide errors can be hard to handle. Bronek Kozicki introduces a C++ library that chains results and errors.

A C++20 functional programming library, **libfn**, expresses mathematical structures and their composition through C++ types, extending the vocabulary available in the standard library.

The library operates on two payload types and four computation carriers:

- **pack**: Product type containing all fields; a tuple-like data structure.
- **copack**: Canonical coproduct containing exactly one alternative; a variant-like disjoint set of types.
- **optional**: Computation yielding a value or empty.
- **expected**: Computation yielding success or error.
- **choice**: Never-failing computation holding one of several alternatives.
- **just**: Never-failing computation yielding a single value or a **void**.

A monadic carrier wraps a value in a computational context, such as possible failure. Carriers can be combined with operations such as **transform** (mapping), **transform_error** (error mapping), **and_then** (sequential monadic binding), **or_else** (recovery), **discard** (terminate sequence), **operator&** (conjunction/simultaneous product composition), **operator|** (disjunction/simultaneous sum composition), and the n-ary folds **fn::conjoin** and **fn::disjoin**. Elimination is **apply** (multidispatch). Pipeline operations which combine a value or an error wrapped in a carrier with a user-supplied function are called *combinators*. In the list above, these are the functor forms of **transform**, **transform_error**, **and_then** and **or_else**. Users may also define their own combinators.

Member vs. pipeline syntax

Some operations are exposed in two forms:

- **Member functions** (for example, **.transform()**, **.and_then()**, **.apply()**) called directly on a carrier or payload (e.g., **ex.transform(f)**, **cp.apply(f)**).
- **Combinators** in namespace **fn** (for example, **fn::transform**, **fn::and_then**) applied via **operator|** (for example, **ex | fn::transform(f)**).

The combinators **recover**, **fail**, **filter**, **inspect**, **inspect_error**, and the pipeline operation **discard** have no member spelling. The member functions **apply**, **apply_r**, **apply_type** and **apply_type_r** have no pipeline spelling.

The **operator|** carries two meanings, told apart by its right operand: a pipeline functor on the right feeds the carrier into that operation, while another carrier on the right is disjunction (explained in Section 7).

Freestanding **fn::apply(f, args...)** is the general multidispatch entry point: it accepts any mix of scalars, tuple-like structures, **packs** and **copacks**, unpacking products and dispatching over alternatives in a single call. Do not confuse with **pfn::apply**, which is a polyfill for the C++26 **std::apply**, meant for C++20 compilers. The **fn::apply** is an extension on top of **pfn::apply**.

In prose, this document omits prefixes (using **apply**, **transform**, **and_then**, **expected**, **pack**) when referring to both forms or core vocabulary types generally.

Storage shape vs. call shape

Although different types can behave identically during application, they remain strictly distinct in memory. For example, **pack<A, B>**, **std::tuple<A, B>**, and **std::pair<A, B>** all unpack into the same call shape **f(a, b)** during **apply**, but they are separate C++ types with distinct layouts. Application does not silently convert or unify types on the storage side.

To illustrate these concepts, the examples in this document use a reusable set of value and error types:

```
struct Error {};
struct OtherError {};

struct A {};
struct B {};
struct C {};
struct D {};

struct UserId {};
struct User {};
struct FilePath {};
struct MaximumSize {};
struct BlockSize {};

struct NotANumber {};
struct OutOfRange {};
struct Missing {
    auto operator<=>(Missing const &) const
        = default;
};
struct IoError {};
struct BadSyntax {};
struct UnknownKey {};
```

1. Why compose types as well as values?

In idiomatic C++, error handling usually means picking one application-wide error enumeration, a giant **std::variant**, or throwing exceptions. If a function only ever fails due to one specific error, returning a large application-wide error variant discards the precise bounds of what the function can actually do.

Bronek Kozicki developed his lifelong programming habit at the age of 13, teaching himself Z80 assembler to create special effects on his dad's ZX Spectrum. BASIC, Pascal, Lisp and a few other languages followed, before he settled on C++ as his career choice. In 2005, he moved from Poland to London, and promptly joined the BSI C++ panel with a secret agenda: to make C++ more like Lisp, but with more angle brackets.

Products and sums of C++ types form an algebra for composing computations.

```
auto parse_id(std::string_view) ->
    fn::expected<UserId, fn::copack<NotANumber>>;
auto validate(UserId) -> fn::expected<UserId,
    fn::copack<OutOfRange>>;
auto load(UserId) -> fn::expected<User,
    fn::copack_for<IoError, Missing>>;

auto graded_pipeline(std::string_view sv) -> void
{
    auto result = parse_id(sv)
        | fn::and_then(validate)
        | fn::and_then(load);

    // The exact derived error union of the
    // pipeline is recorded in the type:
    static_assert(
        std::same_as<decltype(result),
            fn::expected<User, fn::copack_for<IoError,
                Missing, NotANumber, OutOfRange>>>>);
}
```

Listing 1

With `libfn`, the compiler derives an exact, graded error pipeline. Consider parsing, validating, and loading a user (see Listing 1).

The resulting `expected` statically records that the pipeline yields a `User` on success, or fails with exactly one of `NotANumber`, `OutOfRange`, `Missing`, or `IoError`. This exact union accumulates automatically via `and_then` composition.

What does ‘graded’ mean?

With a fixed error type, an `expected<T, E>` requires inputs and outputs in `and_then` computations to share a common error type `E`. This can require defining a common error union upfront – an otherwise unnecessary coupling of types.

`libfn` supports **graded monads**, which you opt into by using `copack` in a monadic carrier, such as `expected`. A *grade* is a set of alternative types, and *graded composition* combines these sets by union. For `expected`, error alternatives form the grade when chaining computations with `and_then`; value alternatives form the grade when composing recovery operations with `or_else`.

The resulting type is **graded**: it expands (or narrows during mapping) to match the alternatives declared by the participating function signatures. This provides strict static effect tracking with zero boilerplate.

When used on the error side of `expected` (for example, `expected<T, copack<Errs...>>`), grading enables `libfn` to derive the combined error alternatives from the participating computations, without explicitly spelling out their union. The resulting types record the algebraic combinations of success and error alternatives declared by those computations. Section 9 explains grading on both sides, including recovery and widening.

The same type precision extends to computations combined side by side rather than in sequence. Conjunction (`operator&`) evaluates

independent computations and bundles them into a single carrier holding a product (a **pack**) of the successful values over a sum of possible error types (a **copack**); disjunction (`operator|`) is its dual. Sections 6 and 7 cover both.

The two cooperating mechanisms

Behind these precise compiled types are two independent mechanisms that cooperate to derive and eliminate these shapes:

1. **Type algebra** records and normalizes the exact stored C++ types using **pack** and **copack** as you compose operations.
2. **The application protocol** uses **apply** and ordinary C++ overload resolution to unpack those stored values and route them to your functions or lambdas.

To route multiple alternative paths inside **apply**, the library provides `fn::overload`, which fuses unrelated lambdas into a single overload set.

These derived types are the explanation of the library’s design, not an internal template-metaprogramming implementation detail.

2. Types as an algebra: zero, unit, alternatives, and products

`libfn` uses an algebraic vocabulary over types:

- `copack<>` represents **0** (Zero) – an uninhabited type.
- `pack<>` represents **1** (Unit) – a type with exactly one state.
- `copack<A, B>` represents **A + B** (Alternatives) – a coproduct where exactly one alternative is present.
- `pack<A, B>` represents **A × B** (Products) – a type where all fields are present simultaneously.

These states can also be used to express the standard vocabulary types:

- `std::optional<T> ≅ 1 + T` (It is either empty/unit or contains `T`, similar to `copack_for<T, std::nullopt_t>`)
- `std::expected<T, E> ≅ T + E` (It contains either success `T` or error `E`, similar to `copack_for<T, fn::unexpected<E>>`)

The symbol $\cong$ indicates an equivalent state shape (an information-level correspondence), not `std::same_as`. `std::optional<T>` is its own distinct C++ type, but algebraically, it behaves as $1 + T$.

Zero is not unit

In `libfn`’s algebra, zero and unit are strictly separated:

- `copack<>` is uninhabited: you cannot construct it. Algebraically, it is **0**.
- `pack<>` is the one nullary product value. You can construct it via `pack<>{}` or `fn::as_pack()`. Algebraically, it is **1**.

Because `pack<>` exists, applying a callable to it invokes a nullary function. Because `copack<>` is uninhabited, providing a callback over `copack<>` is statically proven to be unreachable code (dead code).

Consider the difference in these carrier states:

Computation	Meaning
<code>expected<T, copack<>></code>	An expected value that cannot fail because its error state is uninhabited.
<code>expected<copack<>, E></code>	An expected value that cannot succeed because its success state is uninhabited.
<code>optional<copack<>></code>	An optional that must be empty , as its value state is uninhabited.
<code>expected<void, E></code>	An expected that yields no value on success , but can fail with <code>E</code> .

Copacks use set semantics

A major feature of `libfn` is `copack`, which forms canonical sets of types. For `copacks`, C++ type identity represents equality of sets, while alternatives are selected directly by type – in contrast to the positional indexing of `std::variant`. Combining `copacks` takes the union of these sets, so shared alternatives appear only once. Use the `fn::copack_for` type alias to flatten, deduplicate, and order the types; spelling `copack` directly requires them to already be in this form. (See Listing 2.)

The laws governing `copack` are:

- **Commutative:** Order of types does not change the resulting set.
- **Associative:** Nesting `copacks` is equivalent to flattening them.
- **Idempotent:** Duplicate types are collapsed into one.
- **Identity:** `copack<>` represents the empty set, that is the union unit (adding `copack<>` changes nothing).

Idempotence means a set cannot carry positional meaning: `copack_for<bool, bool>` is one alternative, `copack<bool>`, not two. Types combined into a `copack` should therefore be strongly typed tag structs or distinct domain objects, never generic primitives whose meaning depends on where they sit. Two *distinct* types that the ordering mechanism cannot distinguish are a different matter: the library rejects them outright via a compile-time assertion rather than silently merging them, guaranteeing that no type is ever silently lost.

The algebra is strictly opt-in

The library relies on explicit consent. It does not silently reinterpret arbitrary C++ types as products or coproducts:

- `std::tuple` remains a standard tuple.
- `std::variant` does not acquire `copack` set semantics.
- Tuple-like participation in `apply` changes call shape applicability, but it does not change the stored type identity.

```
auto test_copack_set_semantics() -> void
{
    using SetA = fn::copack_for<NotANumber,
        OutOfRange>;
    using SetB = fn::copack_for<Missing,
        OutOfRange>;

    // Flattening, deduplication, and reordering
    // happen automatically:
    using Union = fn::copack_for<SetA, SetB,
        Missing, Missing, fn::copack<IoError>>;

    static_assert(std::same_as<Union,
        fn::copack_for<IoError, Missing, NotANumber,
        OutOfRange>>);
}
```

Listing 2

- A plain `expected<T, E>` does not automatically become graded.

To invoke the algebra, you use the opt-in mechanisms provided by the library:

- Direct construction of `pack` and `copack_for`.
- Explicit conversions via `fn::as_pack` and `fn::as_copack`.
- Member helpers for explicit type lifting (detailed in Section 9).

If a side is already a `copack` or `pack`, forwarding it behaves naturally without nesting. A `copack` on the error side of `expected` enables error-set unioning (i.e. error grading). Because monadic operations introduce no grade themselves, `and_then` widens a graded error side but rejects a differing ungraded one. Section 9 explains the promotion rules.

The role of void

In `libfn`, `void` is not a member of the payload type algebra – it cannot be stored in a `pack` or a `copack`. Instead it contributes *behaviour*, which can be described as ‘no value and no type’. Hence, `just<void>` and `expected<void, E>` use it to express success without a payload.

In conjunction, a payload-free success contributes no field to the product, as `pack<>` would. In disjunction, that success must remain a possible outcome – if both operands contribute `void` then the resulting value type is `void`. Otherwise, the success without a payload is represented in the resulting sum as `pack<>`. When mapping a `copack`, a `void` result contributes `pack<>`, either on the value or error side.

3. The computation carriers

To model computation and manage control flow (success, failure, alternatives, and empty states), `libfn` uses **computation carriers** (often called ‘monadic types’). The three carrier templates are `optional`, `expected` and `just`. The list below groups them by their fallibility and payload.

The fallible carriers

- `optional<T>` (representing $T + 1$): A carrier that either holds a successful value of type `T` or is empty (`std::nullopt`).
- `expected<T, E>` (representing $T + E$): A carrier that either holds a successful value of type `T` or an error of type `E`.

`expected` is infallible when its error side is `copack<>`.

(Note: `optional` and `expected` are the `fn` extensions of the standards-conforming `pfn` polyfills; Section 15 covers the two layers.)

The infallible (identity) carriers

- `just<T>`: Always contains a single successful value of type `T`.
- `choice<Ts...>`: Always contains one of selected alternatives. It is a type alias to `just<copack<Ts...>>`.

Because `choice` always holds a selected alternative, `choice<>` is incomplete and cannot be constructed: there are no alternative types, hence no active alternative.

Additionally, the infallible state `expected<T, copack<>>` (representing $T + 0 \cong T$) can never fail because `copack<>` represents the initial object `0` (the uninhabited type). Lacking any possible error alternatives, it acts as an infallible, graded unit context. Since it is a specialized state of `expected` rather than a unique template, it is classified under the same computation carrier.

Together, `just`, `choice` and `expected<T, copack<>>` form the **identity cluster**.

Carriers have control flow; raw data does not

Raw algebraic constructs – such as `std::tuple` (product), `std::variant` (sum), or `libfn`’s own `pack` and `copack` – are

passive data layouts. They contain no intrinsic control flow, no concept of short-circuiting, and no built-in notion of success versus failure.

Composing computations requires wrapping these values in computation carriers. Product composition (conjunction) operates on both carriers and data payloads (**pack** and **copack**). When the left operand is either **pack** or **copack**, the right operand can also be a plain value. Sum composition (disjunction) operates only on carriers. The carrier manages success propagation and short-circuits failures.

Carrier bridging: interoperable pipelines

Because these carriers represent different computational contexts, pipelines often need to transition between them. **libfn** provides **cross-carrier bridging** via combinators using **operator|**.

Identity carriers bridge on the success path, via the **and_then** combinator. Since there is no error, the conversions between identity carriers can be freely performed; they can also bridge to fallible carriers (one way only). Section 10 covers that together with the identity cluster.

Standard fallible carriers can bridge to each other on their error/empty recovery paths via the **or_else** combinator (for example, **expected** to **optional**, or vice versa). On success, the value is preserved by a short-circuit; on failure, the user provided recovery callback can convert a missing value into a concrete error, or replace a detailed error with an empty state (see Listing 3).

Member functions **.or_else** and **.and_then** (see Sections 8 and 10) do not support cross-carrier bridging to avoid coupling between different carriers. Combinators, lying a layer above carriers, can perform these transitions without additional coupling.

4. The sum and product payloads: **pack** and **copack**

Modeling complex algebraic structures – such as multi-field products or multi-alternative disjoint sums – requires specialized payload types. **libfn** provides two core vocabulary types.

pack: all fields are present

A **pack** stores multiple fields and supports the standard C++ tuple protocol (**get**, **tuple_size**, **tuple_element**, structured bindings) and an **append** mechanism. Unlike standard tuples, **libfn** packs are strictly flat: a **pack** cannot be an element of another **pack**. Appending a **pack** splices its fields into the outer pack rather than nesting it.

To explicitly lift scalar values into a **pack** (which is useful when conjoining scalars with other packs or copacks), use **fn::conjoin(...)** or **pack{x} & ...** – both hold scalar arguments by value. To store a reference use **fn::as_pack(x)** without template parameters – in this form it is deduction-only and preserves the value category of its arguments. Hence **as_pack(x)** on an lvalue **int x** yields **pack<int&>**, while **as_pack(42)** yields **pack<int>**.

Spelling the template parameters instead (e.g., **as_pack<bool, int>(x, d)**) takes deduction out of the picture: each argument is passed as the type you named, enabling implicit conversions to happen at the call boundary. A reference element becomes something you ask for explicitly – **as_pack<int const&>(x)** yields **pack<int const&>**. Note that partial template spelling is not supported; all element types must be spelled out explicitly if template parameters are specified. (See Listing 4).

copack: one exact alternative is present

As a payload, a **copack** models a discriminated union of types.

Evaluating a **copack** via **.apply()** passes the active alternative to the callback. Because **copack** is self-flattening, nested **copacks** do not occur. A selected alternative that is itself tuple-like – such as **pack**, **std::tuple**, or **std::array** – is unpacked one level, passing its immediate constituents as separate arguments. Since normalized shapes are sums of products, one level of unpacking is sufficient to supply the product's fields as function arguments.

```
auto test_failure_bridge(fn::expected<int,
IoError> ex, fn::optional<int> opt) -> void
{
    // Fallible carriers can bridge to each other
    // on the failure/empty recovery path
    auto expected_to_optional =
        ex | fn::or_else([](IoError) {
            return fn::optional<int>{}; });
    static_assert(std::same_as<decltype(
        expected_to_optional), fn::optional<int>>);

    auto optional_to_expected =
        opt | fn::or_else([]() {
            return fn::expected<int, IoError>{100}; });
    static_assert(std::same_as<decltype(
        optional_to_expected),
        fn::expected<int, IoError>>);
}
```

Listing 3

To explicitly lift a single scalar value into a single-alternative coproduct, use **copack{value}** or **as_copack(std::move(value))**; to store a reference, use **as_copack(value)**. When a **copack** contains *exactly one* alternative type, it is **singular**. It supports direct value extraction both via the index-less **get** and via the tuple protocol (including structured bindings). Direct **get** extraction is **not provided** for multi-alternative

```
auto test_pack(int x = 12, double d = 3.14)
-> void
{
    fn::pack p{UserId{}, User{}}; // CTAD
    [[maybe_unused]] auto [id, user] = p;
    // Structured bindings work naturally

    // Ordered, non-deduplicated fields
    using P = fn::pack<UserId, User, User>;
    static_assert(std::tuple_size_v<P> == 3);

    // Found via ADL (like std::get)
    using std::get;
    static_assert(std::same_as<decltype(get<0>(p)),
        UserId &>);

    // Splicing scalars or other packs via append:
    auto row =
        fn::pack{UserId{}}.append(FilePath{});
    auto wider =
        std::move(row).append(fn::pack{true, 3});

    static_assert(std::same_as<decltype(wider),
        fn::pack<UserId, FilePath, bool, int>>);

    // Explicitly lifting a single scalar value
    // into a pack:
    auto lifted_lvalue = fn::as_pack(x);
    static_assert(std::same_as<decltype(
        lifted_lvalue), fn::pack<int &>>);

    auto lifted_rvalue = fn::as_pack(42);
    static_assert(std::same_as<decltype(
        lifted_rvalue), fn::pack<int>>);

    // Spelling the element type explicitly can be
    // used to opt out of reference preservation:
    auto copied = fn::as_pack<int>(x);
    static_assert(std::same_as<decltype(copied),
        fn::pack<int>>);
    // ... or to force a specific reference type
    // (subject to parameter binding rules):
    auto referenced = fn::as_pack<int const &>(x);
    static_assert(std::same_as<decltype(referenced),
        fn::pack<int const &>>);

    // The explicit form also coerces – the
    // argument converts at the call boundary:
    auto coerced = fn::as_pack<bool, int>(x, d);
    static_assert(std::same_as<decltype(coerced),
        fn::pack<bool, int>>);
}
```

Listing 4

copack types because the active alternative is a run-time property, meaning a multi-alternative **get** would have no single static return type.

A **pack** can be lifted into a **copack** (including **packs** holding references), but a **pack** cannot contain a **copack**. There is algebraic equivalence between a hypothetical **pack** containing a **copack** (which is disallowed) and a specific shape of **copack** containing a **pack** – see Section 6 for details. A **copack** holds a **pack** only by value. (See Listing 5.)

A key safety guarantee of **copack** is **exhaustive matching**. Operations that evaluate a **copack** (such as **transform** and **apply**) delegate to a multidispatch implementation that enforces compile-time exhaustiveness. If the callback or overload set fails to handle any possible alternative in the **copack**, program compilation will fail.

5. Mapping values and errors

Mapping changes contained data without altering the computation’s structural success or failure shape. **libfn** uses **transform** (functor *map*) for the successful channel, and **transform_error** for the error channel. See Listing 6.

Key principles of mapping:

- **transform** preserves the carrier family.
- Success and error states are preserved.
- A bare **copack** has a member **transform** to map across alternatives, but takes no pipeline functor (it’s not a carrier).
- Heterogeneous results inside **transform** or **transform_error** form a normalized **copack** (grading, Section 9).
- Applying **transform_error** to a carrier with no error side (such as **just** or **choice**) is ill-formed.
- When a side is uninhabited (**copack<>**), transformation is well-formed but vacuous: the callback over **copack<>** is neither invoked nor instantiated. This applies to all of **optional<copack<>>**, **expected<copack<>, E>**, and **expected<T, copack<>>**.
- A **void** result is represented in a **copack** as **pack<>**.
- Mapping from **copack** never yields a plain type – even when all branches return the same type, the result remains **copack**.

6. Product composition with operators (conjunction)

Conjunction over carriers combines two independent computations into a single carrier. **a & b** succeeds only if both operands succeed, in which case their *values* combine into a product. If either operand fails, the result retains the leftmost error; for **expected** carriers, the result’s error type is a **copack** (a graded error type), unless both operands share the same plain error type.

```
auto operator_and_composition(
    fn::expected<int, Error> a, fn::expected<bool,
    OtherError> b) -> void
{
    auto result = a & b;
    static_assert(std::same_as<decltype(result),
        fn::expected<fn::pack<int, bool>,
        fn::copack_for<Error, OtherError>>>);
}
```

Functorial action on the initial object

In **libfn**, **transform** implements the functorial map (*fmap*). For the initial object 0 – the uninhabited **copack<>** – there is exactly one map $0 \rightarrow U$ for each chosen target *U*. There is no input value on which to invoke a callback. Haskell expresses this through **absurd :: Void -> a**.

For an uninhabited side, **libfn** preserves the input carrier type without inspecting the callback or requiring it to be callable. This is a library convention consistent with initiality. Section 10 describes the same convention for **or_else**.

```
struct IntegerToken {};
struct StringToken {};

auto test_copack() -> void
{
    static constexpr fn::copack_for<IntegerToken,
        StringToken> token = IntegerToken{};

    // Member apply eliminates the copack by
    // routing the active alternative to an
    // overload set:
    static constexpr auto value
        = token.apply(fn::overload{
        [] (IntegerToken) { return 1; },
        [] (StringToken) { return 2; }});
    static_assert(value == 1);

    // Storing a pack inside a copack is allowed,
    // including a pack holding a reference:
    auto cpr = fn::as_copack(
        fn::as_pack<int const &>(value));
    static_assert(std::same_as<decltype(cpr),
        fn::copack<fn::pack<int const &>>>);

    // Singular lift and direct value extraction
    // (only allowed for singular copacks):
    auto cp = fn::as_copack(42);
    using std::get;
    static_assert(std::same_as<decltype(get(cp)),
        int &>);
}
```

Listing 5

The runtime semantics are:

- The result type records every error either operand can produce; at runtime it holds at most *one* of them – the leftmost failing operand’s.
- Both operands are fully constructed before **operator&** runs, because C++ evaluates operands eagerly. This is an error-selection rule, not short-circuiting: the operator makes nothing lazy or parallel.

Conjunction over data

Unlike disjunction (Section 7), **operator&** applies also to payload types – that is plain types, **pack** and **copack**. Conjunction – also known as product composition – combines values from both operands. When either operand is **copack**, it performs a Cartesian distribution, yielding a **copack** of **packs**. Otherwise, it produces a **pack**, concatenating data from both.

This distribution underpins the rule in Section 4 that a **pack** cannot contain a **copack**. The disallowed nested shape – a product over a sum, $(A \times B) \times (C + D)$ – is algebraically equivalent to $(A \times B \times C) + (A \times$

```
auto mapping_values_and_errors() -> void
{
    fn::expected<UserId, fn::copack_for<Missing,
        IoError>> ex{};

    auto mapped_val = ex | fn::transform([] (UserId)
        { return UserId{}; });
    static_assert(
        std::same_as<decltype(mapped_val),
        fn::expected<User, fn::copack_for<IoError,
        Missing>>>);

    auto mapped_err = ex
        | fn::transform_error(fn::overload{
        [] (Missing) { return BadSyntax{}; },
        [] (IoError e) { return e; }});

    static_assert(
        std::same_as<decltype(mapped_err),
        fn::expected<UserId,
        fn::copack_for<BadSyntax, IoError>>>);
}
```

Listing 6

$B \times D$), a flat **copack** of **packs**. This sum-of-products normal form is enforced by the library – demonstrated in the second case below. (See Listing 7.)

Conjunction **operator&** dispatches on the left operand. To apply it to plain types, lift the left operand, for example as **pack{a} & b**.

The n-ary fold **fn::conjoin(...)** operates in two modes:

- If all arguments are computation carriers, it folds them as a monadic conjunction, equivalent to cascading **operator&**.
- If no arguments are carriers, it conjoins them as a data-level product.

Mixing carriers and data in a single call is ill-formed.

Conjunction with the identity cluster

An operand from the identity cluster (Section 10) contributes a value but never a failure:

- **Unchanged Errors:** Because identity cluster operands never fail, they add no alternatives to the error channel. A **just** or **choice** operand preserves the fallible operand's error side (plain or graded). An **expected<T, copack<>>** operand contributes its empty grade to the error union: no active alternative is added, although the resulting error channel is promoted to a graded **copack**.
- **Value Bundling:** The identity cluster operand's value conjoins with the fallible operand's value into a product.
- **Void Elision:** **just<void>** and **expected<void, copack<>>** act as the product's identity unit and are elided from the value product (for example, **expected<int, E> & just<void>** yields **expected<pack<int>, E>**).
- **Choice Distribution:** Conjoining a **choice** with a fallible carrier distributes the coproduct through the product, yielding a **copack** of **packs** wrapped in the fallible carrier, subject to void elision.

See Listing 8.

7. Sum composition with operator| (disjunction)

Disjunction evaluates alternative computations, keeping the first successful result. **a | b** fails only if both operands fail: dual to conjunction, their values union into a **copack**, and their **errors** combine into a product. If both operands share the same payload type, the value side is unchanged. Otherwise, a carrier with a value type **void** contributes a payload-free success, represented by **pack<>** in the sum.

Strict monoidal structure and distribution

Conjoining **pack** payloads via **operator&** models a monoidal category $(C, \otimes, I)$, with **pack** as the tensor $\otimes$ and **pack<>** as the unit I .

- **Strictness:** because **packs** are normalized flat, there are no distinct-but-isomorphic spellings for the coherence maps to mediate between. $(A \otimes B) \otimes C$ and $A \otimes (B \otimes C)$ are *the same C++ type*, as are $I \otimes A$, $A \otimes I$ and A (when both operands are **pack** types, since a plain type may be automatically lifted). The associator and unitors are therefore identities, and the structure is **strict** rather than merely coherent.
- **Symmetry:** for type union, order does not matter – canonical ordering makes $A \oplus B$ and $B \oplus A$ the same **copack** type. For the product, order matters: a **pack** keeps its fields ordered and unduplicated, so $A \otimes B$ and $B \otimes A$ may yield distinct types related by a genuine swap.
- **Distributivity:** the tensor distributes over the type union $\oplus$ (represented by **copack**), yielding $A \otimes (B \oplus C) = (A \otimes B) \oplus (A \otimes C)$. This is the Cartesian distribution of **pack** over **copack** implemented statically by **libfn**.

If either error side is graded, the product distributes over it: $(E_1 + E_2) \times F \rightarrow (E_1 \times F) + (E_2 \times F)$ (the full Cartesian product when both are graded), yielding a canonical **copack** of **packs**.

```
auto operator_or_composition(fn::expected<int,
    Error> a, fn::expected<bool,
    OtherError> b) -> void
{
    auto result = a | b;
    static_assert(std::same_as<decltype(result),
        fn::expected<fn::copack_for<int, bool>,
        fn::pack<Error, OtherError>>>);
}
```

The runtime semantics are:

- The result type records every combination of failures; at runtime the leftmost operand holding a value wins.
- Both operands are fully constructed before **operator|** runs, because C++ evaluates operands eagerly. This is a value-selection rule, not a lazy fallback.

Disjunction over carriers only

Unlike conjunction, disjunction has no data-level form. Neither **operator|** nor the n-ary fold **fn::disjoin(...)** accepts a **pack**, a **copack**, or a scalar.

```
auto test_cartesian_distribution(fn::copack_
for<A, B> ab, fn::copack_for<C, D> cd) -> void
{
    // Cartesian distribution of copacks: (A + B)
    // x (C + D) = (A x C) + (A x D) + (B x C) +
    // (B x D)
    auto result1 = ab & cd;
    static_assert(
        std::same_as<decltype(result1),
        fn::copack_for<fn::pack<A, C>,
        fn::pack<A, D>, fn::pack<B, C>,
        fn::pack<B, D>>>);

    // Cartesian distribution of a pack and a
    // copack: (A x B) x (C + D) = (A x B x C) +
    // (A x B x D)
    constexpr fn::pack<A, B> Pab = {A{}, B{}};
    auto result2 = Pab & cd;
    static_assert(
        std::same_as<decltype(result2),
        fn::copack_for<fn::pack<A, B, C>,
        fn::pack<A, B, D>>>);
}
```

Listing 7

```
auto test_conjunction_with_identity_
cluster(fn::expected<int, Error> ex,
fn::just<double> j) -> void
{
    // Conjoining an expected with a just
    auto res1 = ex & j;
    static_assert(std::same_as<decltype(res1),
        fn::expected<fn::pack<int, double>, Error>>);

    // Conjoining with a unit (just<void>)
    // elides it, and the value becomes a pack
    auto res2 = ex & fn::just<void>{};
    static_assert(std::same_as<decltype(res2),
        fn::expected<fn::pack<int>, Error>>);

    // Conjoining a choice causes distribution
    // inside the carrier
    fn::choice<bool, double> ch = 1.5;
    auto res3 = ex & ch;
    static_assert(std::same_as<decltype(res3),
        fn::expected<fn::copack_for<fn::pack<int,
        double>, fn::pack<int, bool>>, Error>>);
}
```

Listing 8

Disjunction with the identity cluster

When an operand belongs to the identity cluster (Section 10), the disjunction cannot fail:

- **Error Annihilation:** The error side gains an uninhabited factor (`copack<>`), which annihilates the error product and collapses the channel.
- **Infallible Folding:** The result folds into a non-failing `just` carrier, including when an operand is `expected<T, copack<>>`. A result of type `just<copack<Ts...>>` can also be spelled `choice<Ts...>`.

See Listing 9.

8. Sequential composition

Sequential composition chains dependent operations: `and_then` (monadic *bind*) supplies successful results to the next operation, while `or_else` continues on failure.

The callable passed to `and_then` is a *Kleisli arrow*; it maps successful input to a monadic carrier of the same kind. For an infallible input, the `and_then` combinator also permits the callable to return a different carrier family (carrier bridging, explained in Section 10). In contrast, the `or_else` combinator allows bridging between fallible carriers (Section 3).

Chaining *bind* operations forms a *sequence*, as in the pipeline introduced in Section 1. A sequence may also be graded (explained in Section 9), for example:

```
auto parse_numeric() -> fn::expected<UserId,
    fn::copack<NotANumber>>;
auto load_user(UserId) -> fn::expected<User,
    fn::copack<Missing>>;

auto sequential_bind() -> void
{
    auto result = parse_numeric()
    | fn::and_then(load_user);
    static_assert(
        std::same_as<decltype(result),
        fn::expected<User, fn::copack_for<Missing,
        NotANumber>>>);
}
```

Because member `.and_then` cannot change carrier families (Section 3), its *Kleisli arrow* must return the same carrier family:

- `optional` binds to `optional`.
- Plain `expected<T, E>` binds to `expected<U, E>` (retaining error `E`) or, via singular lift, to `expected<U, copack<E>>` (transitioning to a graded context, Section 9).
- Error-graded `expected` forms the union of heterogeneous error sets with a `copack`.

```
auto test_disjoin(fn::expected<int, Error> a,
    fn::expected<bool, OtherError> b) -> void
{
    // Multiple fallible operands compose cleanly
    auto res1 = fn::disjoin(a, b);
    static_assert(std::same_as<decltype(res1),
        fn::expected<fn::copack_for<int, bool>,
        fn::pack<Error, OtherError>>>);

    // Because just<double> cannot fail, the
    // disjunction with infallible operands becomes
    // infallible
    auto res2 =
        fn::disjoin(a, b, fn::just<double>{1.5});
    static_assert(std::same_as<decltype(res2),
        fn::choice<bool, double, int>>);
}
```

Listing 9

Annihilation and infallibility

Disjunction exchanges the roles of union and product used by conjunction: values add along $\oplus$ where conjunction multiplies along $\otimes$, and errors multiply where conjunction unions.

Under conjunction, `copack<>` is the unit of the error union, $E \cup 0 = E$, so an identity operand preserves the set of possible errors. Under disjunction it is the **annihilator** of the error product, $E \otimes 0 \cong 0$, so a single identity operand empties the channel outright.

The resulting error side is uninhabited by construction – not merely empty at runtime, but incapable of holding a value – which is what renders the whole disjunction infallible and folds it into the identity cluster.

- Copack-valued inputs join heterogeneous successful branch types into a `copack`.
- Branch convergence preserves the exact type without duplicate union states.
- All-`void` branches join to `void`; in mixed `void` and non-`void` branches, `void` contributes `pack<>`.
- Callback results returning bare values require `transform` rather than `and_then`.

The library formalizes this ‘same-kind’ contract via the `fn::same_kind` concept, which lets generic templates probe whether two carrier types belong to the same monadic family with compatible error sides:

```
static_assert(fn::same_kind<fn::optional<int>,
    fn::optional<User>>);
static_assert(fn::same_kind<fn::expected<int,
    IoError>, fn::expected<User, IoError>>);
static_assert(!fn::same_kind<fn::expected<int,
    IoError>, fn::expected<User, Missing>>);
```

Recovery with `or_else` invokes the callback on failure and preserves successful results. For `expected`, the callback receives the error; for `optional`, it takes no arguments. In graded recovery, successful value alternatives accumulate by union (Section 9).

9. Graded compositions

Error-graded composition derives the exact union of the error alternatives declared by the participating computations. When an outer computation holds a coproduct of successful values, and each value requires a different operation to proceed, `libfn` derives a single, normalized `expected` shape.

Consider a configuration reader whose successful results are: `MaximumSize`, `FilePath`, or `BlockSize`. See Listing 10 (next page).

Two independent joins occur during `and_then`:

1. The successful branch values formed the normalized value `copack`.
2. The existing outer errors (`BadSyntax`, `UnknownKey`) and the new branch errors (`OutOfRange`, `Missing`) formed the normalized error `copack`.

Unioning allows different grades of `expected` to share the same carrier family. Although ungraded `expected<T, E>` requires the identical error type `E` (or its singular lift `copack<E>`) to participate in monadic *bind* – meaning `expected<int, IoError>` and `expected<User, Missing>` are not `same_kind` – any two error-graded `expected` types are `same_kind` regardless of how their error sets differ, as the compiler can always derive their union:

```
static_assert(
    fn::same_kind<fn::expected<int,
    fn::copack<IoError>>, fn::expected<User,
    fn::copack<Missing>>>);
```

`expected` supports `copack` on both its error and value sides; `optional` supports it on its value side. Value joining and error grading

```

auto read_config() ->
  fn::expected<fn::copack_for<MaximumSize,
  FilePath, BlockSize>, fn::copack_for<BadSyntax,
  UnknownKey>>;
auto config_pipeline() -> void
{
  auto validated
    = read_config() | fn::and_then(fn::overload{
    [] (MaximumSize v) {
      return fn::expected<MaximumSize,
      fn::copack<OutOfRange>>{v}; },
    [] (FilePath v) {
      return fn::expected<FilePath,
      fn::copack<Missing>>{v}; },
    [] (BlockSize v) {
      return fn::expected<BlockSize,
      fn::copack<OutOfRange>>{v}; }});
  // The result exactly bounds both the
  // successful paths and the error paths
  static_assert(
    std::same_as<decltype(validated),
    fn::expected<fn::copack_for<BlockSize,
    FilePath, MaximumSize>,
    fn::copack_for<BadSyntax, Missing,
    OutOfRange, UnknownKey>>>);
}

```

Listing 10

are independent: branch values can join while the error side stays plain, as in `expected<copack_for<A, B>, E>`.

Recovery via `or_else` behaves symmetrically. It handles input error alternatives and joins any new errors produced by the recovery branches, while preserving the successful value path. Heterogeneous recovery values require a suitable `copack`-valued input. Any original error handled by a branch is removed from the resulting error type unless a branch explicitly re-returns it. When there are no error alternatives to handle – such as in `expected<T, copack<>>` – there is nothing to recover from: the callback is neither invoked nor instantiated.

During sequential composition, `libfn` derives the promoted type from the `copack` you supply:

- In `and_then` (success binding), a plain error type `E` is promoted to `copack<E>` if the returning **error type** of the callback is `copack<E>`.
- In `or_else` (recovery/error binding), a plain success type `T` is promoted to `copack<T>` if the returning **success type** of the callback is `copack<T>`.
- For a **void** success type in `or_else`, the corresponding lift is `copack<pack<>>`.

An ungraded computation thus enters a graded composition without manual lifting.

If you need to perform this promotion explicitly on the carrier itself before entering a composition, `libfn` provides direct member helpers:

- `.copack_error()` on `expected` explicitly lifts the error, transforming `expected<T, E>` to `expected<T, copack<E>>`.
- `.copack_value()` on `expected` explicitly lifts the success value, transforming `expected<T, E>` to `expected<copack<T>, E>`.
- For `expected<void, E>`, the result is `expected<copack<pack<>>, E>`.
- `.copack_value()` on `optional` symmetrically lifts the value, transforming `optional<T>` to `optional<copack<T>>`.

These helper methods provide a compact, explicit alternative to the automatic promotions. See Listing 11.

Widening is subeffecting

A narrow error set can be widened during composition, but narrowing requires explicit handling. *Implicit* narrowing (without

```

auto test_explicit_lifting(fn::expected<User,
  IoError> result, fn::optional<User> opt) -> void
{
  // Explicitly lift the error side of expected:
  auto graded_err =
    std::move(result).copack_error();
  static_assert(
    std::same_as<decltype(graded_err),
    fn::expected<User, fn::copack<IoError>>>);

  // Explicitly lift the value side of expected:
  auto graded_val =
    std::move(result).copack_value();
  static_assert(
    std::same_as<decltype(graded_val),
    fn::expected<fn::copack<User>, IoError>>);

  // Explicitly lift the value side of optional:
  auto graded_opt =
    std::move(opt).copack_value();
  static_assert(
    std::same_as<decltype(graded_opt),
    fn::optional<fn::copack<User>>>);
}

```

Listing 11

handling the removed errors) is unsafe and rejected by the compiler: `transform_error` explicitly remaps error alternatives and requires the callback to handle every input alternative (exhaustive matching of types). However, several alternatives may map to one common error type – a property which can be used for *explicit* narrowing.

Graded monads

In the pomonoid-graded formulation used here, a graded monad is a family of endofunctors indexed by a partially ordered monoid, equipped with unit and binding operations satisfying graded identity and associativity laws.

In `libfn` that pomonoid is carried by the finite sets of C++ types:

- **Grades** (E): Finite sets of alternative types (errors).
- **Monoidal multiplication** ($\cdot$): Set union ($\cup$), representing effect accumulation.
- **Identity** (I): The empty set ($\emptyset$), representing no alternatives on the graded side.
- **Partial order** ($\leq$): Subset relation ($\subseteq$), which licenses effect approximation (subeffecting/widening).

Taking union as the multiplication makes this monoid commutative and idempotent – a join-semilattice – whose order is the one it induces, $E \leq F \Leftrightarrow E \cup F = F$. Those are exactly the `copack` laws of Section 2: widening directly to a given superset agrees with widening through intermediate subsets of that same target.

For a standard monad M , the binding operation maps $M\langle A \rangle \rightarrow (A \rightarrow M\langle B \rangle) \rightarrow M\langle B \rangle$. Write $G_E\langle A \rangle$ for an `expected` with value type A and error alternatives E . For graded success binding with `and_then`, error grades combine by union:

$$\text{bind } G_E\langle A \rangle \rightarrow (A \rightarrow G_F\langle B \rangle) \rightarrow G_{E \cup F}\langle B \rangle$$

Conversely, *recovery* with `or_else` binds errors and accumulates successful value alternatives by union. Write $H_V\langle X \rangle$ for an `expected` with value alternatives V and error type X . Recovery then has the form:

$$\text{recovery} : H_V\langle X \rangle \rightarrow (X \rightarrow H_W\langle Y \rangle) \rightarrow H_{V \cup W}\langle Y \rangle$$

For success binding, the unit injects a value into an `expected` with no error alternatives. For recovery binding, the unit injects an error into an `expected` with no successful value alternatives. In both cases, the unit uses the empty grade, which contributes no alternatives during composition.

The bottom error grade is `copack<>`:

```
template <typename T>
using cannot_fail_t = fn::expected<T,
fn::copack<>>;
```

This computation cannot fail, but it is algebraically prepared to widen if later composition introduces possible errors.

A concrete example of this is `expected<void, copack<>>` (aliased as `fn::expected_unit` in the library). It has one payload-free success state and no error states. It therefore represents the unit type.

In practice, `expected<void, copack<>>` acts as an entry point for graded pipelines. Initiating a chain with this value trigger opts subsequent `and_then` bindings into graded error-set unioning without requiring mock starting errors or manual wrapping. Because its starting error set is empty (`copack<>`), unioning it with subsequent steps' errors (such as `IoError`) yields exactly those errors. An alternative unit type without an error channel is `just<void>` (Section 10).

10. The identity cluster

Certain operations behave like an identity functor across carriers. Because some states correspond structurally, `libfn` licenses specific cross-carrier behavior to avoid boilerplate.

The identity cluster consists of three infallible carrier forms:

Carrier	Algebraic State Shape
<code>just<T></code>	<code>T</code> (A single value)
<code>choice<Ts...></code>	<code>Ts...</code> (A coproduct of values)
<code>expected<T, copack<>></code>	<code>T + 0</code> $\cong$ <code>T</code> (A value and an uninhabited error)

These carriers add no failure or empty state to the outcomes represented by their payloads.

Because no member can hide failure, pipeline bind operations can transition across these boundaries:

```
auto test_identity_cross() -> void
{
    fn::just<UserId> j{UserId{}};

    // Cross-carrier pipeline bind to another
    // identity carrier
    auto result = j | fn::and_then([](UserId u) {
        return fn::expected<UserId, fn::copack<>>{u};
    });
    static_assert(std::same_as<decltype(result),
        fn::expected<UserId, fn::copack<>>>);
}
```

The `bind` operation adopts the carrier family of the provided callback – a crossing only the pipeline-scoped functors are licensed to make (Section 3).

Success-path bridging

Fallible carriers (excluding `expected<T, copack<>>`) and `optional` cannot transition to infallible carriers, because doing so would risk silently discarding an active error or empty state.

In contrast, identity carriers can bridge to any fallible carrier via pipeline `fn::and_then`. Because identity carriers are statically infallible, transitioning to `optional` or a standard `expected` merely introduces potential downstream failure. No pre-existing failure is discarded, as none can exist upstream. See Listing 12.

All three cluster members (`just`, `choice`, and `expected<T, copack<>>`) can bridge to fallible carriers.

Monadic operations on the identity cluster:

- **Success Mapping (transform):** Preserves the carrier family. A `copack` returned from a callable mapped over a `just` produces a `just` of that `copack`, which is the `choice` over its alternatives (Section 11).

```
auto test_success_bridge(fn::just<int> j) -> void
{
    // An identity carrier can bridge to fallible
    // carriers on the success path
    auto to_opt = j | fn::and_then([](int i) {
        return fn::optional<int>{i}; });
    static_assert(std::same_as<decltype(to_opt),
        fn::optional<int>>);

    auto to_exp = j | fn::and_then([](int i) {
        return fn::expected<int, IoError>{i}; });
    static_assert(std::same_as<decltype(to_exp),
        fn::expected<int, IoError>>);

    // Bridging a multi-alternative choice to
    // fallible optional with heterogeneous success
    // join:
    auto choice_to_opt
        = fn::choice_for<int, bool>{true}
        | fn::and_then(fn::overload{
            [](int) -> fn::optional<char> {
                return {'a'}; },
            [](bool) -> fn::optional<long> {
                return {2L}; }); });
    static_assert(std::same_as<decltype(
        choice_to_opt),
        fn::optional<fn::copack_for<char, long>>>);
}
```

Listing 12

- **Sequential Binding (and_then):** Allows cross-carrier transitions within the identity cluster (such as `just` to `expected<U, copack<>>`) when using pipeline `fn::and_then`.
- **Recovery & Error Mapping (transform_error, or_else, recover, inspect_error):** Because `just` and `choice` lack an error side, these are ill-formed. On `expected<T, copack<>>`, they are vacuously well-formed but the error callback is statically unreachable.
- **Short-Circuiting (fail, filter):** Ill-formed for identity carriers, as no failure (error or empty state) can be constructed from an infallible context.
- **Elimination Fallbacks (value_or):** Ill-formed on `just` and `choice`. On `expected<T, copack<>>`, it remains well-formed to support generic code, but the fallback must still initialize `T` even though its branch is statically unreachable. If the value side is `void`, no fallback is accepted.
- **Neutral Observation (inspect, discard):** Supported and behave normally.

11. choice: identity over a coproduct

`choice<Ts...>` is an infallible carrier whose operations dispatch to the active alternative. `choice{x}` copies scalar value `x` into the single-alternative `choice`, or a `choice` over its alternatives if `x` is a `copack`. To store a reference to scalar `x`, use `fn::as_choice(x)`. Empty `choice<>` is an incomplete type.

Mapping into a choice

`transform` on a `just` whose callable returns a `copack` yields a `choice` – the same identity functor, its payload now a coproduct. See Listing 13 (next page).

Similarly, member `and_then` on a `just` can return a `choice` directly. Pipeline `fn::and_then` also permits a transition to `expected<T, copack<>>`.

A callback passed to member `and_then` must return a `just`, including `just<void>` or a `choice`. Choice branches may return different `just` types. If those types agree after cv/ref removal, the result keeps that type; otherwise their payloads form a normalized choice. An ordinary `just<T>` contributes `T`, a returned choice contributes its alternatives, and `just<void>` contributes `pack<>`, as a `void` side does in a disjunction.

```

auto test_identity_transformation(
    fn::just<UserId> j) -> void
{
    // Transforming a just with a callable
    // returning a copack produces a choice
    auto mapped
        = j | fn::transform([](UserId) {
            return fn::copack_for<Missing,
                FilePath>{Missing{}}; });
    static_assert(std::same_as<decltype(mapped),
        fn::choice_for<FilePath, Missing>>);
}

```

Listing 13

Inside its domain, **choice** behaves differently from bare **copack**:

- **copack** is self-flattening data; a **copack** returned from a branch dissolves into the result.
- **choice** is an atomic, never-failing computation; a **choice** returned from a branch remains an alternative unless **and_then** joins it.

Consider a scenario where different branches of a switch return different **choice** types (Listing 14).

A callback returning a bare value requires **transform** rather than **and_then**; **choice**'s **and_then** rejects bare value returns with a compile-time diagnostic.

12. Elimination and multidispatch

Eliminating computation structures yields an ordinary C++ value. This is typically achieved via **apply** or **apply_r**. Singular **copack** supports direct extraction via **get** (Section 4), and **just** supports total value extraction via **.value()**. On fallible carriers, **.value()** is partial: it yields the value if present, and otherwise throws (**bad_expected_access**, **bad_optional_access**).

transform and **apply** differ in whether the structure survives:

- **transform** remains inside the carrier or **copack**, producing a new carried type.
- **apply** eliminates the structure: the result type is deduced from the branches, which must all return the same type.
- **apply_r<R>** permits any branch results that convert to **R**.

Application expands one selected level only. The call shapes are straightforward:

Type	Eliminated Call Shape
A	f(A)
pack<A, B>	f(A, B)
pack<>	f()
std::tuple<A, B>	f(A, B)
copack<A, B>	f(A) or f(B)
copack<pack<A, B>, C>	f(A, B) or f(C)

A whole-carrier **expected** application cleanly handles both success and error paths into one result type:

```

auto test_elimination(fn::expected<UserId,
    fn::copack<Missing>> ex) -> int
{
    return ex.apply(fn::overload{[] (UserId) {
        return 1; }, [] (Missing) { return 0; }});
}

```

Exhaustiveness is statically constrained. If no handler can accept a possible input, the compilation fails. **fn::overload** is merely a helper; final selection always relies on ordinary C++ overload resolution.

Type-tagged elimination

Because storage shape and call shape are distinct, untagged **apply** can erase structural context (such as in **expected<int, int>**). To

Why copack always flattens, but choice does not

Categorically, **copack<Ts...>** represents a **coproduct** payload (disjoint sum $\oplus T_i$), whereas **choice<Ts...>** places that payload in an identity-monad carrier. These two entities have different purposes, hence also different behaviour:

1. copack normalizes the payload algebra:

Normalization deliberately removes grouping between sums (for example, $(A + B) + C \cong A + B + C$): a nested **copack** is ill-formed by design. The constraint is intentional: it allows **transform** on **copack** to collapse every branch's results into a flat, deduplicated set instead of an ever-nesting type.

2. choice preserves the carrier boundary:

To distinguish mapping from monadic composition, the monadic carrier **choice<Ts...>** wraps the sum in an 'identity layer' to preserve structural depth: **choice<choice<T>> ≠ choice<T>**. That layer holds eager flattening in check.

Since **choice<Ts...>** is an alias for **just<copack<Ts...>>**, the underlying identity-wrapper model is $M(A) = \text{just}(A)$ with A being the coproduct payload in the case of **choice**. In this model, the unit and multiplication correspond to wrapping and unwrapping the carrier:

- **Unit/return** $\eta_A : A \rightarrow M(A)$: wraps the coproduct in the **choice** layer. Building a **choice** from a single alternative composes this with the coproduct's own injection $\iota_i : T_i \rightarrow \oplus_j T_j$.
- **Join/flatten** $\mu_A : M(M(A)) \rightarrow M(A)$: removes one **just** layer.
- **Bind**: maps, then flattens explicitly via **join**. Making that step explicit is what grants control over *when* flattening occurs.

3. When binding combines different returned **choice** types, their payload alternatives are flattened and deduplicated according to the **copack** rules. This determines the resulting payload type, separately from removing the carrier layer.

Warning — catch-all handlers defeat exhaustiveness

Exhaustiveness requires a valid handler invocation for every alternative. An unconstrained **[] (auto&&...)** accepts all alternatives, so a **copack** that later gains an alternative still compiles – the new alternative routes into the catch-all instead of failing where its branch is missing. Where one handler should serve several alternatives, name a concept instead of **auto** – for example **[] (MyConcept auto &&)** – and the exhaustiveness check still fires for anything the concept does not match.

```

auto test_choice_mapping(fn::choice<User, UserId>
    ch) -> void
{
    constexpr auto mapper = fn::overload{
        [] (UserId) { return
            fn::as_choice(Missing{ }); },
        [] (User) { return
            fn::as_choice(FilePath{ }); }};

    // transform nests the returned choice as a
    // mapped value
    auto mapped = ch | fn::transform(mapper);
    static_assert(
        std::same_as<decltype(mapped),
            fn::choice_for<fn::choice<FilePath>,
                fn::choice<Missing>>>);

    // and_then joins and flattens them into a
    // normalized superset choice
    auto bound = ch | fn::and_then(mapper);
    static_assert(std::same_as<decltype(bound),
        fn::choice_for<FilePath, Missing>>);
}

```

Listing 14

Elimination of algebraic structures

The two eliminations are not dual to one another – the dual of coproduct elimination is product *introduction*, pairing – and they differ in how much the category determines.

- **Coproduct**: eliminating $A+B$ is canonical. Supplying $f : A \rightarrow C$ and $g : B \rightarrow C$ determines a *unique* mediating morphism $[f, g] : A+B \rightarrow C$ by the universal property, and in **libfn** ordinary C++ overload resolution is what computes it.
- **Product**: its universal property characterizes maps **into** $A \times B$; the operation $A \times B \rightarrow C$ must be supplied by the caller. What **apply** supplies is the bridge between the product as *stored*, a **pack**, and the product as an argument list, which C++ keeps distinct.

Carrier elimination (**apply_type**) additionally preserves the injections: the state tag – such as **std::in_place**, **fn::unexpected** or **std::in_place_type<T>** – tells the handler *which* injection morphism placed the value into the structure.

preserve context and prevent implicit conversions from conflating different states, **libfn** provides **apply_type** (and **apply_type_r**) member functions.

Eliminating a carrier using **apply_type** passes an explicit state tag as the first argument to the handler, followed by the unpacked payload:

- **expected**: The success arm receives **std::in_place** and the success value (or **std::in_place** alone if **void**), while the error arm receives **fn::unexpected** and the error.
- **optional**: The success arm receives **std::in_place** and the value, while the empty arm receives **std::nullopt**.
- **copack** and **choice**: The active alternative arm receives **std::in_place_type<T>** and the payload.
- **just**: The arm receives **std::in_place_type<T>** and the value. Symmetrically, **just<void>**'s arm receives **std::in_place_type<void>** alone, representing success without a payload.

13. The monadic operations map

A reference of **libfn** operations, organized by channel and effect:

Success channel

- **transform**: Maps the successful value. Stays inside the carrier.
- **and_then**: Sequences success-path computations. The mechanism for introducing new errors into a graded expected.
- **filter**: Enters a short-circuit state if a predicate fails.
- **inspect**: Observes the successful value transparently.
- **fail**: Intercepts success and forces a transition to a failure state.

Error/empty channel

- **transform_error**: Maps the error value. Stays inside the carrier and can narrow a graded error set (Section 9).
- **or_else**: Sequences computations based on errors. Joins recovery values.
- **recover**: Intercepts failure and forces a transition back to a success state.
- **inspect_error**: Observes the error value transparently.
- **value_or**: Supplies a fallback. Member **.value_or(x)** eliminates the carrier to return the value; pipeline **fn::value_or(x)** preserves the carrier, returning it engaged with either its own value or the fallback.

Neutral

- **discard**: Discards the carrier and returns **void** to signal that the return value is deliberately ignored.

Elimination

- **apply**: Routes the stored state to an overload set, leaving the algebra with an ordinary C++ value.
- **apply_type**: The same elimination, keyed by an explicit state tag.

Composition & combination

- **operator&** (conjunction): Combines independent computations (values into a **pack**, errors as a union).
- **operator|** (disjunction): Combines alternative computations (values into a **copack** disjoint sum, errors as a product **pack**).
- **fn::conjoin**: An n-ary fold of **operator&**, over monadic carriers or over packs, copacks and scalars – not a mix of the two.
- **fn::disjoin**: An n-ary fold of **operator|** over monadic carriers, supporting total disjunction with the identity cluster.

Key architectural rules of the map

- **fail and recover are duals**: **fail** transitions success to failure ($Success \Rightarrow Failure$), and **recover** transitions failure to success ($Failure \Rightarrow Success$). Neither operation widens a graded error set.
- **Error-graded and_then**: Combines input and callback error alternatives by union of types.
- **Value-graded or_else**: Similarly combines input and callback value alternatives.
- **filter and fail**: Enter an existing short-circuit state; the carrier must already support the failure state.
- **Error-side operations** (such as **transform_error**, **or_else**, **recover**, and **inspect_error**): Require a carrier with an error or empty side. They are ill-formed on **just** and **choice**, and remain vacuously well-formed on **expected<T>**, **copack<>>** (Section 10).

14. Laws as C++ equalities

When results can be compared in a constant expression, instances of the laws can be checked by the compiler. The following checks illustrate functor identity and monad left identity (Listing 15, next page).

Other properties hold:

- **Functor composition**:

```
m | transform(f) | transform(g)
```

equals

```
m | transform([](auto&&... v) {
    return g(f(v...)); })
```

(for singular parameters; for packs, use **apply**).

- **Monad right identity**: **m | and_then(pure)** equals **m**.

- **Monad associativity**: **(m | and_then(f)) | and_then(g)** equals:

```
m | and_then([](auto v) {
    return f(v) | and_then(g); })
```

For graded expected, both sides of the associativity derive the same normalized union grade.

- **Product associativity**: Holds after canonical **pack** normalization.
- **Coproduct set semantics**: Union associativity, commutativity, and idempotence apply.

```
constexpr auto test_laws() -> void
{
    constexpr fn::expected<int,
        fn::copack<Missing>> ex{42};

    // Functor Identity: mapping with identity
    // yields the same value
    constexpr auto id = [](auto v) { return v; };
    static_assert((ex | fn::transform(id)) == ex);

    // Monad Left Identity: pure(x) >= f is
    // equivalent to f(x)
    constexpr auto pure = [](int v) { return
        fn::expected<int, fn::copack<Missing>>{v}; };
    constexpr auto f = [](int v) { return
        fn::expected<int,
            fn::copack<Missing>>{v * 2}; };
    static_assert((pure(42)
        | fn::and_then(f)) == f(42));
}
```

Listing 15

- **Coherent widening:** Upcasting an error through intermediate supersets yields the same final type as upcasting directly to the broadest superset.

15. C++ mechanics that preserve the algebra

To ensure reliability, **libfn** uses compiler mechanisms to reject malformed usage and preserve performance.

Constraints and exhaustiveness

Public concepts and **requires** clauses constrain which calls are accepted. **fn::monadic_invocable** verifies pipeline invocation, while concepts such as **fn::applicable_transform** and **fn::applicable_and_then** express individual applicability rules. These checks enforce exhaustive handling of alternatives where a callback is invoked and require complete type definitions to be present. This underpins the compile-time exhaustiveness guarantees of **apply** and monadic operations established in Sections 4 and 12.

One normalization order per program

The **copack** normal form rests on a strict total order over types, and the default and **LIBFN_CXX26** modes may derive different orders (Section 2). Each mode therefore places **fn** in its own ABI namespace: the mode becomes part of every **fn** symbol, so the two modes never link as one, and **copacks** whose layouts disagree cannot merge unnoticed. (**pfn** carries no such split – it has no normal form to protect.)

C++ value properties

The library respects C++ value mechanics:

- Core operations are **constexpr**.
- **const** is propagated.
- The algebra's own types (**pack**, **copack**, **just**, and **choice**) are structural when their elements are, so a **constexpr** value of these types may be used as a template parameter.
- Value categories (lvalue/rvalue) propagate strictly to callbacks.
- Reference collapsing rules apply to carriers and their payloads.
- Immovable and move-only payloads are supported in place.
- Reference-bearing **pack<T&...>** and **copack<T&...>** are supported. Lifetime management of non-owning references remains with the caller.

- **pack** compares element-wise, supporting equality and three-way comparison. For reference-bearing **pack<T&...>** and **copack**, comparison applies to the referents.

pfn and fn

The library is divided into layers:

- **pfn** (Polyfill **fn**) is the standards-facing layer. It provides **std::optional** and **std::expected** in their C++26 shape – monadic member functions, **optional<T&>**, range support – plus smaller utilities such as **std::invoke_r** and **std::unreachable**, all available to a C++20 compiler.
- **fn** is the strict extension layer. It introduces the **pack/copack** algebra, multidispatch, graded monads, **choice**, **just**, the pipeline functors, and the composition operators **&** and **|**.

Every **fn** type with a **pfn** counterpart is a superset of it. Mixing **pfn** and **fn** within a single program is allowed, provided that types from the two namespaces **do not interact**. For example, storing an **fn::pack** in a **pfn::optional** is *not supported*, but the two may be used *independently* in the same program. Switching a valid program from **pfn** to **fn** is not expected to change compilation nor program behaviour. ■

Functional terminology

For readers with a background in functional languages (like Haskell or OCaml), this table translates standard terminology to **libfn**'s C++ vocabulary:

Functional term	libfn equivalent
fmap / map	transform / transform_error
bind / >>=	and_then
pure / return	just<T>{v} / expected<T, copack<>>{v} – a carrier constructor
Lift / inject	fn::as_pack / fn::as_copack / fn::as_choice
Kleisli arrow	The callable passed to and_then
Product type	pack / std::tuple
Coproduct / Sum	copack (the sum itself) / choice (the never-failing carrier over a sum)
Subeffecting	Widening an error grade / subset inclusion

Further reading

For the mathematical background to the structures modeled in **libfn**, refer to:

1. Orchard and Petricek, 'Embedding effect systems in Haskell' (for effect sets, union, and subeffecting) available at: <https://www.doc.ic.ac.uk/~dorchard/publ/haskell14-effects.pdf>
2. Orchard, Wadler, and Eades, 'Unifying graded and parameterised monads', specifically Definition 21 (for the graded-monad interpretation) available at <https://arxiv.org/pdf/2001.10274>
3. McDermott and Uustalu, 'Flexibly Graded Monads and Graded Algebras', available at: <https://dylanm.org/flexibly-graded-monads.pdf>

Note: **libfn** does not claim to fully implement their flexibly graded construction, but the work contextualizes graded structures.

4. Emily Riehl, 'Category Theory in Context' (for the categorical foundations, including universal properties, products, coproducts, and monads), available at: <https://emilyriehl.github.io/files/context.pdf>

The library is available at github.com/libfn/functional

Load-Resilient C++ Servers

Building predictable servers is hard. Ajay Pandey uses `select()`, bounded queues, and modern C++ concurrency to solve the problems.

A real-time server can be stable under moderate loads but fail when requests suddenly arrive faster than the server can process them. Incoming connections keep on coming, queues fill up, delays rise, and memory usage increases. The process may even stay alive, but the quality of service becomes much worse from the user's point of view. The issue is often not just a lack of computational resources, but the lack of a clear strategy for coping with overload. This also results in higher latency, disruption of services, losses and a tarnished corporate reputation due to loss of customers.

This article discusses a simple and elegant architecture of a C++ server capable of dealing with load spikes. This server uses a non-blocking I/O thread, bounded queues, worker threads, and backpressure control. Instead of trying to increase the throughput to the maximum possible level, the architecture aims to provide predictable latencies, bounded resource consumption, clear ownership of the network I/O, and graceful degradation in case of overload.

Introduction

Consider a RESTful API that consistently demonstrates reasonable response times and resource utilization. One day, however, due to a software release, an automated retry storm or an accidental increase of traffic volume, the rate of incoming requests temporarily surpasses the processing capacity of the server. Connections are still accepted, queues start growing, response times skyrocket from milliseconds to seconds, and memory starts being used up. Even though the process is apparently alive and well, the users are experiencing increasing latency and failure of their requests.

The problem is seldom about insufficient processing capacity; most of the time it is about the inability to have a coherent approach to handling requests once the system is overwhelmed. Servers accepting requests indefinitely will shift the overload from the network level to the level of the application memory, increasing queueing delay, tail latency, and recovery difficulties. An overloaded system might look like a working one even if it does not deliver acceptable service anymore.

Sometimes, a very simple architecture might solve the problem quite effectively. This architecture would consist of a single non-blocking I/O thread with `select()` [Linux-7], a bounded queue for incoming requests, a worker thread pool for CPU-heavy or blocking application logic, and an output queue going back from the workers to the I/O thread.

It allows a good separation of concerns while still maintaining simplicity. Instead of maximizing the throughput, the goal is to maintain predictable latency, bounded memory usage and graceful degradation in case the load is higher than the processing capacity of the server.

Despite the common perception that `select()` is obsolete, its limitations are often not well understood. Instead of imposing some limitations on the request rate, the limitation of `select()` lies in the traditional `fd_set` interface with its `FD_SETSIZE` value and the need to constantly rebuild and scan the sets of descriptors. The costs of doing that are quite negligible when the server is handling a relatively small number of concurrent connections, thus `select()` is an excellent example of architecture that can easily be applied to `epoll`, `kqueue`, `Boost.Asio` and even other modern event-driven frameworks. [Linux-2] [OpenBSD] [Boost]

One of the most powerful methods to control the overload is to make buffering explicit. Instead of having unlimited queue, the bounded queue makes the application choose how it needs to handle overload. Depending on the workload, the decision could be either to immediately drop requests, admit them with a delay, use rate limiting, shed some of the low priority tasks or use TCP flow-control features. What is important about this approach is that overload is becoming an explicit engineering choice, not an unintentional effect of buffering. [IETF-5]

Once the queue is bounded, however, another question arises: how large should that bound be? An arbitrary capacity may prevent unlimited memory growth, but an excessively large queue can still produce unacceptable waiting times. Queue capacity should therefore be related to the application's latency objectives and expected workload. We return to this question in the Backpressure and Sizing section.

Correctness of this architecture relies on understanding how the underlying OS works. On Linux, the accepted sockets are not guaranteed to inherit `O_NONBLOCK` property, which is why [Linux-1]

```
accept4(..., SOCK_NONBLOCK | SOCK_CLOEXEC)
```

should be used as the default setting. Also, `select()` modifies its descriptor sets and, on Linux, its timeout value on each call. Similarly, `recv()` and `send()` may process only part of the data or return `EAGAIN` status code. Closing a monitored socket from another thread can lead to undefined behavior. Thus, this naturally leads to the architecture where I/O thread manages the lifetimes of sockets and all the network-related I/O operations, and worker threads are solely responsible for request processing and producing responses.

In the following sections we are going to implement this architecture with modern C++.

Framing the problem and design goals

Processing a high volume of requests is not the problem *per se*. A robust server must continue to operate reliably when its capacity to process incoming work is temporarily surpassed. Under bursty loads, design goals should shift from optimizing for maximum throughput to having

Ajay Pandey is a software engineering leader and distributed systems architect specializing in modern C++, concurrency, and high-frequency event-driven systems. He has over two decades of experience developing C++ backend application servers for Fixed Income trading systems and Intelligent Network platforms in the telecom industry. His interests include scalable architectures, low-latency engineering, event coalescing, and resilient real-time platforms. You can reach him at ajaypandey1@gmail.com or through his LinkedIn profile: <https://www.linkedin.com/in/ajay-pandey-76158310/>

One important aspect of our architecture that is often overlooked is that there are two different paths of communication with completely different concurrency semantics

bounded memory consumption, predictable latency, explicit overload handling capabilities, and sufficient observability to understand where processing time is being spent. Sometimes, a server that rejects excessive work and stays responsive is far more valuable than a server which keeps consuming work until it inevitably fails.

This article uses `select()` to keep the event loop simple and portable, making the architecture easier to explain. However, the same considerations apply to `epoll`, `kqueue`, `Boost.Asio`, and other event-loop frameworks. Please note that `select()` is suitable for relatively small servers only. Linux `epoll` or BSD/macOS `kqueue` serve a similar purpose to `select()`, but scale better when monitoring large numbers of sockets/file descriptors. Now let's look at the key concepts such as concurrency, ownership, and backpressure.

There are several key principles behind this architecture.

First, the **event loop must be light-weight**. Its job is to accept new connections, receive data from the sockets [Linux-10], assemble complete request frames, dispatch work to the bounded queue, collect completed responses, and write responses back to the clients. Event loop is fully responsible for managing the lifetime of connections and doing all network I/O. This avoids many synchronization issues and eliminates races caused by concurrent socket reads, writes, or closure. The event loop must never do anything expensive like executing any business logic or performing any blocking operations. Shortening the I/O path gives us enough flexibility to keep the server responsive even when the workers are busy.

Second, we want **networking and application logic to be separated**. Request processing is performed by a worker pool using modern C++ concurrency primitives such as `std::jthread`, cooperative cancellation with `std::stop_token`, and RAII-based thread lifetime management. Workers are only dealing with application data and never touch the socket state directly.

Third, **admission must be bounded**. Accepting each new request increases the resource utilization (memory consumption, scheduling time, CPU). Accepting new requests forever just shifts the overload from network onto application memory where it grows in queueing delay until latency becomes unacceptably high. Bounded queue forces us to take an explicit decision whenever we run out of capacity like whether to reject new requests, to admit them with delay, to apply some kind of rate-limiting or to drop less important work. Overload becomes a controlled state of our application rather than an accidental outcome of the unlimited buffering.

Finally, **cooperative cancellation and observability** are important parts of our design. They allow workers to finish their work without accepting any new requests, and expose metrics at each step of request processing. Waiting in the queue, processing time, total latency, queue size and request rejection rates give us much more information about the health of our server than its throughput, helping us to spot overload early on.

One important aspect of our architecture that is often overlooked is that there are two different paths of communication with completely different concurrency semantics. Our request queue has **SPMC (single-producer multiple-consumer)** model: the event loop adds requests to the request queue, and multiple worker threads process them. The response queue, on the contrary, uses **MPSC (multiple producer single consumer)** model: multiple workers add completed responses to the response queue, and the event loop retrieves them and sends them to the client's sockets.

While special queue implementations may optimize both communication patterns separately, the reusable bounded queue constructed with the help of modern C++ synchronization primitives such as `std::mutex`, `std::condition_variable`, and RAII locking often turns out to be the most convenient solution providing a great balance between simplicity and correctness. As the server grows, a single event-loop thread may eventually become a bottleneck, requiring a multi-loop architecture. However, multi-loop design is outside the scope of this article. A reusable queue abstraction makes such future changes easier without requiring a redesign of the overall architecture.

At the end of the day, our goal is not to build the fastest server but the most reliable and observable one. These features do not come from any specific I/O multiplexing API but from explicit ownership, bounded resources, clearly defined concurrency boundaries, and overload policies.

System architecture overview

The overall architecture is intentionally simple, with each component responsible for a single well-defined task, as illustrated in Figure 1 (next page). Incoming connections are managed by a single event loop built around `select()`. The event loop parses incoming requests, places them onto a bounded work queue, and later delivers completed responses back to clients. All application-specific processing is delegated to a pool of worker threads, while operational metrics are collected throughout the request lifecycle.

The most important design decision is **ownership**. The event loop is the sole owner of every connection. It is responsible for accepting new connections, reading incoming data, managing socket state, writing responses, and closing connections. Worker threads never interact with sockets directly; they operate exclusively on parsed requests and produce completed responses. This clear separation significantly reduces synchronization complexity and eliminates an entire class of race conditions associated with concurrent reads, writes, and connection teardown.

All sockets are created in **non-blocking mode**. On Linux, `accept4()` is the preferred mechanism because it allows newly accepted sockets to be created atomically with both `SOCK_NONBLOCK` and `SOCK_CLOEXEC`. Relying on `accept()` followed by `fcntl()` introduces an unnecessary race window, and accepted sockets do not reliably inherit the `O_NONBLOCK` flag from the listening socket. Configuring non-blocking behavior at creation time results in a simpler and more robust implementation. [Linux-1] [Linux-4]

While lock-free queues are frequently used in performance-oriented applications, their use comes at the cost of increased complexity in memory ordering, reclamation, and correct operation.

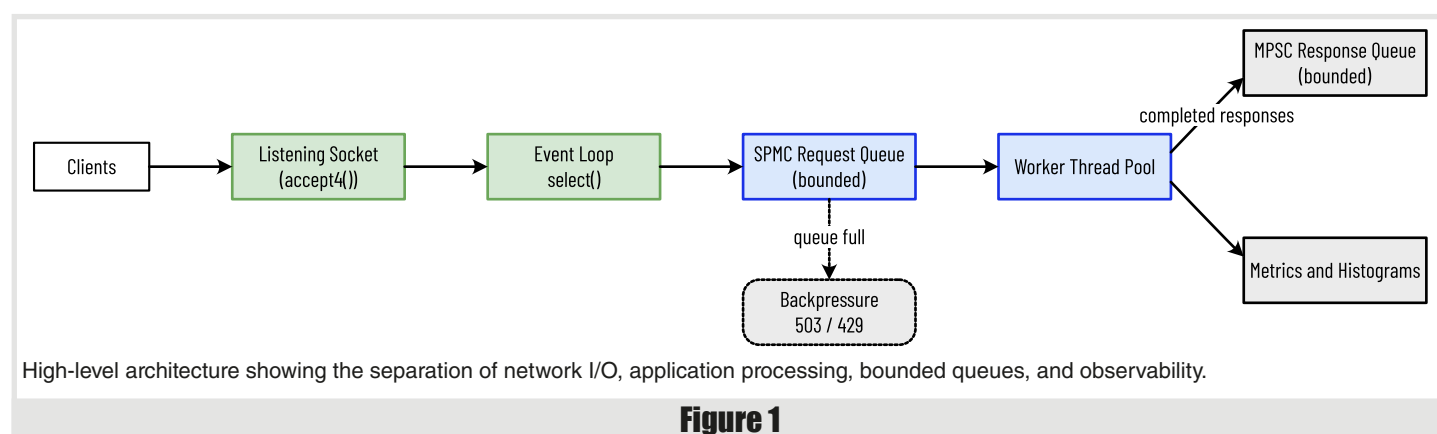


Figure 1

With non-blocking sockets, operations such as `recv()` and `send()` may complete only partially or return `EAGAIN` (or `EWOULDBLOCK`) when no progress can be made immediately. Rather than treating these conditions as errors, the event loop simply resumes processing other connections and retries the operation when the socket becomes ready again. This event-driven model allows a single thread to manage many concurrent connections without blocking on any individual client. [Linux-6] [Linux-8]

A natural consequence of this design is that **all network writes remain centralized within the event loop**. This decision is motivated by correctness rather than performance. Stream-oriented sockets make no guarantee that a call to `send()` will transmit the entire response, and preserving response ordering while handling partial writes quickly becomes difficult if multiple worker threads are permitted to write directly to the same socket. By routing completed responses through a bounded response queue, the event loop becomes the single writer for every connection, ensuring ordered delivery while greatly simplifying synchronization and buffer management.

The architecture also benefits from treating the event loop as the central coordination point for the entire server. Besides monitoring client sockets, it can observe additional file descriptors that represent internal events. On Linux, an `eventfd` provides an efficient mechanism for waking the event loop from another thread, for example, during shutdown or configuration changes. `timerfd` integrates periodic timers and connection timeouts into the same event-driven infrastructure. Monitoring these descriptors through the existing `select()` loop avoids ad hoc synchronization mechanisms and eliminates the need to interrupt the event loop by closing sockets from another thread, a behavior that POSIX leaves unspecified and that Linux documentation explicitly discourages relying upon. [Linux-3] [Linux-12]

Although the implementation presented in this article uses `select()` for clarity, the overall architecture is intentionally independent of any particular I/O multiplexing mechanism. Changing the I/O multiplexing mechanism affects the event-notification layer, but not the fundamental design. The fundamental design remains unchanged: a lightweight event

loop owns every connection, bounded queues regulate the flow of work, worker threads execute application logic, and explicit ownership keeps synchronization predictable and maintainable.

Implementation of core components

Now that we have the architecture in place, the implementation of other parts is much easier. Implementation includes **three parts** that cooperate with each other: **a bounded queue** to manage the workload, **a worker pool** that processes the workload and **an event loop** that manages all connections. All the parts have clear responsibilities, thus simplifying the development process and testing as well.

The initial implementation will be done with simplicity and correctness in mind rather than optimizing for maximum performance. Modern C++ includes very efficient mechanisms for synchronization, which are quite good enough for server applications. Complexity can be added only when measurements show if it is needed.

The modern bounded queue

The bounded queue provides the concurrency barrier between the event loop and the worker pool. Its role is equally simple yet vital like transferring the work from one thread to another and limiting the amount of work the application can accept.

While lock-free queues are frequently used in performance-oriented applications, their use comes at the cost of increased complexity in memory ordering, reclamation, and correct operation. It is usually better to first ask the simpler question: **does profiling reveal that the queue is the problem area?**

In most cases, the performance bottleneck of the system lies elsewhere for example, be it the application itself, database operations, other services, or network latency. A properly designed bounded queue using standard synchronization mechanisms in C++ is thus an excellent start.

Modern C++ allows such an implementation to be both brief and clear. The bounded queue uses `std::mutex`, `std::condition_variable`, RAIL-style locking, and `std::optional` to allow blocking consumers,

Equally important is what the workers do not do... Their only job is to transform requests into responses

bounded producers, and graceful shutdown of the application. The bounded queue also has another meaning besides serving as a synchronization primitive: **the bounded queue defines admission boundary of the application**. Once the queue gets full, some kind of decision has to be made: whether to reject new work, to defer admission, to apply backpressure, or to shed lower priority work. By making overload visible, queue helps to build resiliency of the server rather than simply acting as a transport between threads.

The implementation in Listing 1 is an example of bounded queue implemented using modern C++ synchronization primitives.

The implementation intentionally omits lock-free techniques for now. Should profiling later demonstrate that queue contention has become a measurable bottleneck, the implementation can be replaced without affecting the surrounding architecture because the queue already provides a clean abstraction boundary.

Building the worker pool

After requests go into the bounded queue, the job is done by the worker pool exclusively. The workers take requests from the queue, process an

application-specific handler, and put responses in the response queue to transmit via the event loop.

There is a modern C++ solution to traditional thread pools created using `std::thread` in the form of `std::jthread` that has the automatic joining on destruction feature and the cooperative cancellation mechanism provided by `std::stop_token`. (See Listing 2.)

Equally important is what the workers do not do. Worker threads do not access sockets directly. They do not call `recv()` and `send()`, and they do not close client connections. Their only job is to transform requests into responses. This separation significantly simplifies synchronization and allows the networking and application logic to evolve independently.

The response queue is naturally **multiple-producer, single-consumer (MPSC)** topology because there are many producers (worker threads) and one consumer (the event loop).

The event loop

The event loop contains most of the correctness problems in the design of network servers. Luckily, having some simple ownership rules makes the implementation much easier.

The event loop owns each socket during its whole lifecycle. It accepts new connections, takes data from them, reconstructs whole request

```
template<typename T>
class BoundedQueue
{
public:
    explicit BoundedQueue(std::size_t capacity);

    [[nodiscard]]
    bool try_push(T value);

    template<class Rep, class Period>
    [[nodiscard]]
    bool push_for(
        T value,
        std::chrono::duration<Rep, Period> timeout);

    [[nodiscard]]
    std::optional<T> pop();

    [[nodiscard]]
    std::optional<T> try_pop();

    void close();

    [[nodiscard]]
    std::size_t size() const;

private:
    std::deque<T> queue_;
    mutable std::mutex mutex_;
    std::condition_variable not_empty_;
    std::condition_variable not_full_;
    std::size_t capacity_;
    bool closed_{false};
};
```

Listing 1

```
struct Request
{
    int fd;
    std::string payload;
    std::chrono::steady_clock::time_point admitted;
};

struct Response
{
    int fd;
    std::string payload;
    bool close_after_write{false};
};

class WorkerPool
{
public:
    using Handler =
        std::move_only_function<Response(Request)>;
    WorkerPool(std::size_t workers,
        BoundedQueue<Request>& requests,
        BoundedQueue<Response>& responses,
        Handler handler);

private:
    void run_worker(std::stop_token token);

    std::vector<std::jthread> workers_;

    BoundedQueue<Request>& requests_;
    BoundedQueue<Response>& responses_;

    Handler handler_;
};
```

Listing 2

It is this clear separation of responsibilities, which makes the architecture resilient

frames, sends them to the bounded request queue, receives completed responses from the worker pool, writes back to clients, and finally closes connections when it is necessary. There is no way for any other thread to change the socket state.

Such an ownership model is very helpful when using non-blocking sockets. Calls to `recv()` and `send()` may process only part of the data or return `EAGAIN` (or `EWOULDBLOCK`) when the operation cannot complete immediately. Such behavior is a regular case of event-driven processing.

Moreover, `select()` requires some additional conditions to be met. First of all, descriptor sets are changed in place and should be reinitialized before each `select()` call. In Linux OS, the same applies to the timeout structure. Otherwise, there will be some hard-to-detect bugs.

With the help of modern C++, this code becomes more readable but does not change anything in the semantics. One could express timeouts with `std::chrono`, manage resources with RAII, use cooperative shutdown with `std::stop_token`.

Listing 3 is the simplified example of the event loop.

Despite being highly oversimplified, this diagram (Figure 1) represents the basic event loop architecture. Its full-fledged implementation is just an expansion of every step with keeping the same ownership rules.

```
using namespace std::chrono_literals;

constexpr auto poll_timeout = 200ms;

while (!stop_token.stop_requested())
{
    rebuild_descriptor_sets();
    drain_completed_responses();
    auto timeout = to_timeval(poll_timeout);
    const int ready =
        ::select(max_fd + 1,
                &read_set,
                &write_set,
                nullptr,
                &timeout);
    if (ready < 0)
    {
        if (errno == EINTR)
            continue;
        throw std::system_error(
            errno,
            std::system_category());
    }
    accept_new_connections();
    read_requests();
    write_pending_responses();
}
```

Listing 3

Why centralized I/O is important

One could try to give permission to every worker thread to write its response to the client socket. Although this might seem as decreasing of queuing overhead, it will make synchronization much more complicated.

The stream sockets don't guarantee that the whole buffer would be sent during one `send()` call. Partial sends happen quite often, especially in high-load conditions. Allowing multiple worker threads to perform sending responses would make it difficult to preserve responses order and to handle partially sent buffers.

Routing of every completed response through the response queue makes the event loop the **single writer** for every connection. The buffer handling, partial sends, connections shutting down, and responses order remain in one module, which makes this architecture more understandable and debuggable.

Event loop cooperation

While observing client sockets' activity, the event loop can watch other events as well. On the Linux platform an `eventfd` can be used to notify the event loop from other thread, while `timerfd` helps to integrate timers and connection timeout handling into the event-driven system.

It allows avoiding the undefined behavior by trying to wake up the `select()` with closing of the monitored descriptor in another thread. The event loop continues to own the lifetime of every socket, while wake-ups and shutting down requests go through synchronization primitives.

Design takeaways

These principles are independent of the particular I/O multiplexing mechanism.

It is this clear separation of responsibilities, which makes the architecture resilient. Event loop takes care of communication, the worker pool takes care of computing, the queues manage the work distribution, and every module has a single responsibility. It leads to creation of the predictable, measurable, and maintainable server.

Backpressure and sizing

Backpressure is not a single mechanism; it is a stack of mechanisms applied at different layers of the system. At the application layer, the server can reject work, defer admission briefly, apply rate limits, or shed lower-priority requests. At the connection layer, it can stop reading from a socket or close the connection. At the transport layer, TCP already provides flow control through receiver windows, retransmission behavior, and zero-window probing.

For HTTP services, overload also has well-defined protocol-level semantics. **503 Service Unavailable** is appropriate when the service as a whole is temporarily overloaded, often with a short **Retry-After** header. **429 Too Many Requests** is better reserved for per-client, per-user, or per-principal throttling. **431 Request Header Fields Too Large**

Overloading must not be an unintended consequence of infinite queues, kernel buffer storage, or blocking invocations

Strategy	What it does	Latency effect	Throughput effect	Best use
Immediate reject	Fails fast when the bounded queue is full; for HTTP, usually returns 503 or 429	Best protection for tail latency	Reduces accepted throughput intentionally	Default policy for interactive APIs
Brief defer	Waits for a tiny bounded interval for queue capacity	Can smooth microbursts, but risks hiding queueing if overused	May slightly increase accepted throughput during short bursts	Small transient bursts
Priority shedding	Drops or refuses low-priority work first	Preserves latency for critical traffic	Keeps useful throughput high	Mixed-priority workloads
Token-bucket throttle	Limits admission rate per client, key, endpoint, or tenant	Improves fairness	Prevents one source from monopolizing workers	Multi-tenant or public APIs
Transport pushback	Stops reading so TCP flow control constrains senders	Moves pressure into kernel and socket state; latency may rise sharply	Can stabilize ingress, but obscures overload	Secondary mechanism, not the primary policy
Connection abort / overflow reset	Closes or resets connections when the service cannot keep up	Harsh but bounded	Preserves server capacity at client expense	Last-resort fail-fast behavior during severe overload

Table 1

is useful when admission should fail before work reaches the application layer because the request itself is too large. [IETF-1]

A practical comparison of overload strategies looks like Table 1 (next page).

A good rule of thumb for most HTTP servers would be the following: immediately drop a connection when the request queue becomes full; respond with the **503 Service Unavailable** status code and a short **Retry-After** header when overloaded globally; use **429 Too Many Requests** for per-client or per-principal overload throttling. [IETF-3]

There are a number of TCP-level tweaks available as well, although they can't be seen as a substitute for proper admission control at the application level. **TCP_NODELAY**, for instance, reduces latency for short responses by disabling Nagle's algorithm [IETF-2]. **TCP_DEFER_ACCEPT** allows to wake up the server only when data comes into the listening socket. **TCP_INFO** exposes a bunch of useful Linux TCP statistics. [Linux-11]

On the other hand, more aggressive TCP tweaks like aborting connections that have grown too long are not a proper alternative to the combination of bounded request queues, quantifiable latency and overload handling policy.

When sizing request queues, the starting point should be the latency goals that the server has to achieve. In particular, the Little's Law [Wikipedia] of queueing theory provides a convenient starting point: $L = \lambda W$, where L is the average queue length, λ is the arrival rate, and W is the average queue waiting time.

When expressed in terms of $W = L/\lambda$, the equation means that the queue size should be set up taking into account both the arrival rate of requests

and how much time the server is ready to spend waiting before serving those requests.

Such a relation proves to be a handy starting point in capacity planning. With a known arrival rate and allowable queueing delay, an initial queue-size estimate can be derived. On the other hand, if the queue capacity is fixed, then the wait will grow linearly with the load.

The above relation is an approximation. Real-life loads tend to be bursty, service times are not constant, and **p95/p99** (95th and 99th percentile) latencies can be substantially higher than the mean. Nonetheless, Little's Law gives a much more defensible starting point compared to 'a few thousands of items just in case'. The queue size can later be tuned through burst tests and adjusted using the measurements of queue waiting time, service time, end-to-end latency and rejection ratio. [Little11]

Similarly, the number of worker threads should be selected taking into consideration the expected service time. If the server spends most of its time waiting for the completion of the tasks on other hosts, its scalability will be quite different from that of CPU-bound processes.

A small overload helper keeps the admission policy explicit (Listing 4, next page).

The essential thing is not the policy itself which is illustrated above, but that there is an explicit policy at all. Overloading must not be an unintended consequence of infinite queues, kernel buffer storage, or blocking invocations. It must be an intended component of the server's behavior, measurable in metrics and discernible in the code.

Measurement, observability, and testing

It is impossible to understand a resilient server by reporting just one number, called latency. There are several distinct phases for processing a

Several failure modes emerge quite often when implementing hand-written `select()`-based servers

```
enum class OverloadPolicy
{
    reject_immediately,
    defer_briefly,
    shed_low_priority
};
[[nodiscard]]
bool admit_or_overload(
    BoundedQueue<Request>& queue,
    Request request,
    OverloadPolicy policy,
    Connection& connection)
{
    using namespace std::chrono_literals;
    auto reject = [&connection]{
        connection.out =
            "HTTP/1.1 503 Service Unavailable\r\n"
            "Connection: close\r\n"
            "Retry-After: 1\r\n"
            "Content-Length: 0\r\n\r\n";
        connection.close_after_write = true;
        return false;
    };
    switch (policy)
    {
        case OverloadPolicy::reject_immediately:
            if (!queue.try_push(std::move(request)))
                return reject();
            return true;

        case OverloadPolicy::defer_briefly:
            if (!queue.push_for(std::move(request),
                               2ms))
                return reject();
            return true;

        case OverloadPolicy::shed_low_priority:
            // In a real system, Request would carry
            // priority, tenant, endpoint, or cost
            // metadata. Low-priority work can then
            // be refused before critical work.
            if (!queue.try_push(std::move(request)))
                return reject();
            return true;
    }
    return reject();
}
```

Listing 4

request that represent different aspects of the server's behavior. Only the measurement of total response time makes it hard to figure out where the latency occurs and to troubleshoot performance problems.

At least, there should be three independent measures of latency:

- **Time in the queue:** the period between receiving a request and starting its processing by a worker.
- **Service time:** time that an application logic executes in the context of the worker thread.

- **Total latency:** total processing time since the moment when a complete request is received until all bytes of the response have been written to the client.

Each measure reveals a different story. The growth of latency in the queue means that requests arrive too quickly. The increase in service time means that either an application takes more time to execute or it calls a slow external component. The end-to-end latency measure shows how the client sees the server.

Modern C++ allows such instrumentation at low cost. A tiny structure is enough to collect all the necessary timestamps:

```
using Clock = std::chrono::steady_clock;

struct Timings
{
    Clock::time_point admitted;
    Clock::time_point worker_start;
    Clock::time_point worker_done;
    Clock::time_point response_sent;
};
// queue_wait   = worker_start - admitted
// service_time = worker_done   - worker_start
// end_to_end    = response_sent - admitted
```

Instead of merely keeping the averages, each measurement needs to be logged into a latency histogram. **HdrHistogram** [HdrH] is a great open-source tool for doing so since it offers fixed memory logging along with percentile queries which make collecting such histograms feasible in high throughput systems.

Failure modes, shutdown, and security

Several failure modes emerge quite often when implementing hand-written `select()`-based servers. The first one is that one may assume that a socket, which was reported as readable or writable, will pass the whole amount of requested data. In reality, `recv()` can return only the available bytes, and `send()` may write only part of the data or return **EAGAIN** on non-blocking sockets. `select()` itself can return **EINTR**, and descriptor sets along with timeouts should always be re-built on each iteration. Handling those situations as part of normal execution flow is crucial for creating solid event loops.

Another common issue is that sockets should never be closed from worker threads, because it's undefined behavior according to POSIX, and Linux does not wake up `select()` at all in that case. Instead, workers should inform event loop about need to close the connection, and I/O thread will handle removing the descriptor, performing proper `shutdown()` and closing the socket using `close()`. Including an `eventfd` into the event loop helps here to wake up event loops from those coordination messages. [Linux-9]

Graceful shutdown should be taken into account in the architecture too. The server should stop accepting new connections, process all outstanding requests, wait until queues become empty or until some shutdown timeout occurs, flush all pending responses, and shut down connections. Sending `shutdown(fd, SHUT_WR)` will ensure that any data will be sent out

before the socket will be closed, and **MSG_NOSIGNAL** flag will prevent unnecessary generation of **SIGPIPE** on writing to already closed socket.

Timeouts should be handled at the level of event loop too, instead of trying to rely on socket-level receive or send timeouts. On Linux, **timerfd** fits naturally into event-driven architecture, and **TCP_USER_TIMEOUT** option will allow to control how long stalled peer connections will be kept alive. Similar way, failures at the level of the listening socket should be fixed by optimizing application-level latency, tuning sizes of queues or increasing processing capabilities of application before trying any TCP-level tuning. [Linux-5]

Security is a concern which starts much earlier than applying TLS to a server. Robust request framing and parsing will help to avoid processing of malformed or ambiguous requests. Request headers should have limits, conflicting **Content-Length** and **Transfer-Encoding** headers should be rejected, oversized requests should fail with appropriate HTTP status code, like **431 Request Header Fields Too Large**. As far as internet-facing services are concerned, TLS should be considered default transport either by the server itself, or by the reverse proxy. [IETF-4]

Recommendations for defaults and production checklist

The proposed architecture deliberately emphasizes simplicity, clarity of ownership, and predictable behavior over excessive optimizations. On Linux, new sockets returned by **accept4()** should have the flags **SOCK_NONBLOCK** and **SOCK_CLOEXEC** set in order to ensure atomicity of both non-blocking mode and close-on-exec semantics. Request and response queues should always have a known finite capacity, which can be calculated on the basis of latency goals of the application. Size of worker pool should be derived from the measurements of service time and adjusted via workload measurements, not request rate. For HTTP-based services, **503 Service Unavailable** together with short **Retry-After** header is a good choice as the default response in the case of global overload of the server, while **429 Too Many Requests** is more suitable for the client-based rate limiting.

It is recommended to use the flag **MSG_DONTWAIT** for keeping the writing process non-blocking and **MSG_NOSIGNAL** to suppress generation of **SIGPIPE** when clients unexpectedly close connections. It is usually easier to manage idle connection deadlines within the event loop than via timeouts on the socket level and thus, the former method allows the timeout mechanism to be integrated into the rest of the server architecture. The evaluation of performance should be based on the tool, which is able to generate requests with constant rate, and it is recommended to record latency separately as queue wait time, service time and end-to-end response time histograms.

Before putting a server implemented according to this architecture into production, it is necessary to check several key assumptions about it. All sockets must be configured for non-blocking operations, and the event loop should rebuild its **select()** descriptor sets and timeouts at the beginning of every iteration. The event loop should be the only owner of socket read/write operations and connection lifetime management, while the worker threads should deal with application data only. The request and response queues should have a known finite capacity, the overload policy should be explicitly specified, and the latency should be measured separately as queue wait time, service time and end-to-end response time.

Request framing should be checked strictly and oversized headers must be rejected properly. Transport security model, in particular, termination point of TLS protocol must be specified – either the application itself or reverse proxy acts as SSL/TLS terminator. Ultimately, implementation details do not matter much in comparison with key principles of the architecture. Keep all queues bounded, make overload evident, provide the single-threaded ownership of connection state, measure queueing latency separately from the service time – those principles will be beneficial for any server regardless of what event loop API is used.

Conclusion

Designing an event-driven server is not just about achieving maximum performance; it is about ensuring consistent behavior once demand begins to exceed processing capabilities. The separation of concerns of network I/O operations and the actual logic, bounded queue sizes, backpressure, and clear delineation of ownership of connections are all necessary elements in developing a highly resilient system. Combined with the capabilities provided by modern C++ concurrency constructs and observability, these practices can result in responsive and measurable systems. While **select()** is used here to demonstrate the concepts, the same architecture can be applied to **epoll**, **kqueue**, and **Boost.Asio**. The technologies and APIs may change over time, but the fundamental principles of limited resources, admission control, cooperative concurrency, and measurement will always be at the core of building resilient event-driven systems. ■

References

- [Boost] Boost C++ Libraries Overview: https://www.boost.org/doc/libs/latest/doc/html/boost_asio/overview.html
- [HdrH] High Dynamic Range Histogram: <https://hdrhistogram.github.io/HdrHistogram/>
- [IETF-1] RFC 6585 – ‘Additional HTTP Status Code’, available at: <https://datatracker.ietf.org/doc/html/rfc6585>
- [IETF-2] RFC 896 – ‘Nagle’s algorithm’, available at: <https://datatracker.ietf.org/doc/html/rfc896>
- [IETF-3] RFC 9110 – ‘HTTP Semantics’, available at: <https://datatracker.ietf.org/doc/html/rfc9110>
- [IETF-4] RFC 9112 – ‘HTTP/1.1’, available at: <https://datatracker.ietf.org/doc/html/rfc9112>
- [IETF-5] RFC 9293 – ‘Transmission Control Protocol (TCP)’, available at: <https://datatracker.ietf.org/doc/rfc9293/>
- [Little11] John D. C. Little ‘Little’s Law as Viewed on Its 50th Anniversary’ in *Operations Research* 59:3, published May-June 2011, available at <https://people.cs.umass.edu/~emery/classes/cmpsci691st/readings/OS/Littles-Law-50-Years-Later.pdf>
- [Linux-1] **accept()**, Linux manual page, available at: <https://man7.org/linux/man-pages/man2/accept.2.html>
- [Linux-2] **epoll()**, Linux manual page, available at: <https://man7.org/linux/man-pages/man7/epoll.7.html>
- [Linux-3] **eventfd()**, Linux manual page, available at: <https://man7.org/linux/man-pages/man2/eventfd.2.html>
- [Linux-4] **fcntl()**, Linux manual page, available at: <https://man7.org/linux/man-pages/man2/fcntl.2.html>
- [Linux-5] **listen()**, Linux manual page, available at: <https://man7.org/linux/man-pages/man2/listen.2.html>
- [Linux-6] **recv()**, Linux manual page, available at: <https://man7.org/linux/man-pages/man2/recv.2.html>
- [Linux-7] **select()**, Linux manual page, available at: <https://man7.org/linux/man-pages/man2/select.2.html>
- [Linux-8] **send()**, Linux manual page, available at: <https://man7.org/linux/man-pages/man2/sendmsg.2.html>
- [Linux-9] **shutdown()**, Linux manual page, available at: <https://man7.org/linux/man-pages/man2/shutdown.2.html>
- [Linux-10] **socket()**, Linux manual page, available at: <https://man7.org/linux/man-pages/man7/socket.7.html>
- [Linux-11] TCP, Linux manual page, available at: <https://man7.org/linux/man-pages/man7/tcp.7.html>
- [Linux-12] **timerfd_create()**, Linux manual page, available at: https://man7.org/linux/man-pages/man2/timerfd_create.2.html
- [OpenBSD] **kqueue()**, OpenBSD manual page, available at: <https://man.openbsd.org/kqueue.2>
- [Wikipedia] Little’s Law theorem: https://en.wikipedia.org/wiki/Little%27s_law

Afterwood

Testing is, of course, important. Chris Oldwood discusses where the line is drawn between being thorough and being obsessed.

One of the first meetups I attended – when we could finally come out of hiding once the pandemic was waning – was DDD London. After the talk on Executable Specifications had finished, we retired to the pub across the road and started chewing the fat. I'd had a slightly frustrating time at work that day, so opened the conversation with a minor rant about a colleague who'd raised a pull request where the code didn't even parse, let alone fail some tests. This wasn't the first time, either; we'd chatted before when I'd noticed them generating pull requests with lengthy commit histories due to typos and silly bugs. This tale generated some sympathetic sighs from my audience, along with some testing related chatter that eventually led to someone posing the question: "When do we become test-obsessed?"

If there's something I learnt from Seb Rose's extended lightning talk back at the ACCU 2013 conference, it's that overly dramatic words have a tendency to come with unfortunate traits. His talk was titled *Are you Passionate?* and it pointed out that job adverts for 'passionate programmers' are effectively asking for people who are highly emotional and potentially reckless! He posited that we should really be asking for the *opposite*, i.e. dispassionate programmers, people who are methodical and considerate. (The 'move fast and break things' crowd would likely disagree, but most of us work for companies who actually have customers and a reputation worth protecting.)

Likewise, if you look up the term 'obsessed', it implies an *unhealthy* fascination, i.e. taking an extreme interest in a subject. I find great benefit in the practice of testing, but I don't like the thought of being considered obsessed by their utility!

Of course, this question wasn't intended literally, it was just a figure of speech, but it did get me thinking about my attitude towards testing in case it might be verging too far towards the wrong end of the scale. I seem to spend more time these days thinking about the idea of 'working software as the primary measure of progress', and how that is realized throughout the delivery pipeline to avoid wasting time building the wrong thing, or reworking badly implemented features. Adopting a Kaizen approach (continuous improvement) means every little mistake *can* be an opportunity to re-evaluate the process and work out how to eliminate that source of waste going forward. There are always trade-offs to consider, but in general software development is about the long game, and that means considering productivity as a function of the entire system's lifetime.

Winding back to what originally triggered this obsession with being potentially obsessed, I don't think anyone would consider it extreme to double-check locally that your code compiles, or in the case of a dynamically typed language, parses. Furthermore, running any automated tests should also be a no-brainer. The build server is great at watching my back, but doing all this locally *first* allows me to get the fastest feedback possible. That stops me context switching and ensures I concentrate on finishing what I've started. I can then get the change into production quicker, and consequently start to get a return on that investment.

Where, I suspect, the waters start to get murky is when considering the effort required to create a suite of fast running, reliable, automated *acceptance* tests. Once you step out of the world of pure functions and start factoring in side-effects and external services, writing those kind of tests starts to get much harder. The 'GOOS' book (*Growing Object-Oriented Software, Guided by Tests*) is over 15 years old now and introduced the 'outside-in' style of development, which depends on being able to write automated end-to-end tests (customer oriented), as well as the more traditional unit tests (programmer oriented). While in-memory mocks seem to have caught on, no doubt because they're easy to write, the real gem – automated acceptance tests – typically remain far more elusive; this is despite the leaps in technology which make testing with external dependencies so much easier now.

Of course, not every change is a simple functional change that can be easily verified with a selection of automated tests, most notably deployment and infrastructure related changes, or where real world data is just too messy to distil into a comprehensive test set. Anywhere it's not obvious from the committed tests *how* the testing was done I like to see some kind of a test plan in the commit message, PR description, or work item ticket. One of the problems with the term 'code review' is that it completely ignores any review of the test approach, which can easily escalate into 'isn't that QA's problem?' I get it... testing is way less fun than writing code, even test code, but fire-fighting means nobody gets to have any fun.

Once a feature is deployed to production, we can move on the next big thing, right? Not so fast! The term 'testing in production' makes some people nervous, so I prefer 'validate in production', i.e. is it actually doing what we expected, where it *truly* matters? There is a technical angle here – did we deploy/apply it correctly – because test environments are always a crude approximation of the real world, but there is also a business one. Observability tries to close the loop between the features we add (or remove) and the business outcome we obtain.

But what if we didn't make it possible to observe the effects of our feature in a useful way? The way to ensure that happens comes right back at the beginning during the refinement process, and, for the delivery team it's likely the first of many times when they get to ask the probing question, "But how do we test this?" (Only after "Should we even build this?", of course.)

Hence, those of us who might be regarded as test obsessed are probably just the ones who feel the need to *keep* asking that awkward question because we've seen too many assumptions make an ass out of us and umption (ask Samuel L. Jackson). You can dress it up as checking, validating, or evaluating, or talk about acceptance criteria and conditions of satisfaction, or customer demos, but it's still all about the various feedback loops, and finding ways to keep them short.

Ultimately the term 'working software' boils down to just *one* test, the acid test: does it solve the customer's problem? Unfortunately, getting to that point can sometimes involve a test of one's patience. ■

Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros. These days it's enterprise grade technology from plush corporate offices the comfort of his breakfast bar. He also commentates on the Godmanchester duck race and is easily distracted by emails and DMs to gort@cix.co.uk and @chrisoldwood



Join ACCU

Run by programmers for programmers,
join ACCU to improve your coding skills

- A worldwide non-profit organisation
- Journals published alternate months:
 - *CVu* in January, March, May, July, September and November
 - *Overload* in February, April, June, August, October and December
- Annual conference
- Local groups run by members

Join now!
Visit the website



professionalism in programming

www.accu.org

accu

professionalism in programming



Monthly journals
Annual conference
Discussion lists

To find out more, visit accu.org