100,000 Lines of C Later: Re-architecting Enterprise Postgres Backups in Go

Jimmy Angelakos
Staff Software Engineer, pgEdge

PGDay UK
London, 2026-09-08

pgEdge®

About Jimmy Angelakos

- Staff Software Engineer, pgEdge. Based in Edinburgh, Scotland
- Involved in the Open Source Community for 25+ years
- PostgreSQL Significant Contributor
- Organizer, PostgreSQL Edinburgh User Group
- Member, PostgreSQL Europe Diversity Committee
- Author, PostgreSQL Mistakes and How to Avoid Them
- Co-author, PostgreSQL 16 Administration Cookbook
- Creator of pgEdge ColdFront (Postgres tiering to Iceberg)
- Creator of pg_statviz (Postgres stats visualization)

What is this talk about?

First ½: Background

- What happened in April 2026
- What a backup tool really does
- “The ten invariants”
- How do you test an invariant?

Second ½: Design

- Three design tenets
- Three operating modes
- Where did 100,000 lines go?
- The parts that cannot shrink

Background

What happened?

- pgBackRest: 1 of the 2 de facto enterprise backup tools for Postgres
  - Stable release: 2016
  - A decade of production lessons incorporated into code
  - Written in Perl
  - Now written in C
- April 2026: corporate backing was lost
  - Maintainer archived the repository ☠️
- A few weeks later, it survived
  - Now maintained again
- For those few weeks, the question was "what do we back up with?"

Why was that a problem?

- 96,000 lines of dense C code
  - And another 80,000 lines of test harness
- Manual memory management, custom networking, its own protocols
- Excellent code, but maintained by very few people
  - Not many people can understand this code and work on it
- Critical infrastructure needs alternatives that more people can maintain...

So I started writing one

Introducing pgSafe

- Written from scratch in Go
- Same concepts & operational rules as pgBackRest
  - None of the code
- Development team
  - Already half as many maintainers as pgBackRest!
  - j/k thanks for all your work David and Stefan ❤️
- Goal: functionality parity for common deployments
  - Full and incremental backups, PITR
  - POSIX, S3, Azure Blob, GCS, SFTP
  - PostgreSQL 13 through 18
- Alpha maturity level! Please don't back up production with it yet

Ground rules

- Read pgBackRest's source to learn the rules
  - Write the Go from the rules, not from the C
- The very best of open source: learning, not copying or freeloading
- Every module is a "sealed box" with a small public interface
- TDD: no code without unit, integration and end-to-end tests
- Every line of code has to justify its existence

“With every modern tool at my disposal”

- This includes AI coding assistance
- So is this vibe-coded AI slop?
- No, because the assistant works under the same rules as I do
  - KISS, DRY, UNIX-style sealed modules, stdlib first, no ORM
  - Non-negotiable TDD: unit, integration, end-to-end, every change
- Architecture, tenets and invariants are owned by me
- Every code decision is reviewed → supervision is the actual job
- The responsibility is mine → that cannot be delegated

“Every modern tool at my disposal” (ii)

- Go
  - Standard library with HTTP, TLS, crypto
  - JSON built in
  - Generics
  - `errgroup`
  - Garbage Collector
- Go’s toolchain
  - `go test`, `golangci-lint`, race detector
  - Cross-compilation, one static binary

What does a backup tool really do?

How does a backup work?

- Tell Postgres a backup is starting
- Copy every file in the data directory somewhere else
- Tell Postgres the backup is finished
- Make sure the WAL from that window is safely archived
- Write down what you copied, with checksums
- Be able to put it all back, at a point in time of your choosing
- Sounds simple, right?
  - The hard part is the ordering

The WAL bracket

- `pg_backup_start()` → copy files → `pg_backup_stop()`
- The files you copied in between are individually inconsistent
  - Postgres kept writing to them while you were reading them
- Replaying WAL from start LSN to stop LSN repairs all of that
- Without that WAL, the backup is worthless
  - Every single segment has to be there
- Everything else in the tool exists to serve this bracket

Three ways to get the WAL

- `archive` (default): `archive_command` copies each completed segment
  - Full PITR reach, needs `archive_mode=on` and a working command
- `stream`: `pg_basebackup --wal-method=fetch` with the WAL included
  - Self-contained, pair it with a WAL archive and PITR works as usual
- `walgrab`: our worker reads `$PGDATA/pg_wal` after stop
  - Same self-contained result, no `pg_basebackup` involved
- Restore doesn't care which one you used

“The ten invariants”

Where do the rules come from?

- Every invariant is a lesson somebody learned the hard way
- They're not implementation details
  - They don't change between releases
- Each one has a name, a reason and a named test
- A regression on any of those tests must block the release
- They are equally valid across all 3 operating modes

#1: Ordering: stop, wait, manifest

- Upload every file and make it durable
  - fsync (POSIX), multipart-complete (object stores)
- Then `pg_backup_stop()`, which gives you the end LSN
- Check the archive until every required segment is there
- Only then you can write the manifest
- Get this wrong and you get a backup that
  - Looks consistent
  - Cannot be restored
- Test: kill the backup between any of the steps

#2: Manifest is the source of truth

- Manifest presence means: this backup exists and is valid
- Written last, and written atomically
  - POSIX: write `manifest.tmp`, fsync, rename
  - Object stores: upload tmp, then conditional copy to final
- Half-written manifest means corruption: `info` will lie to you
  - Retention will delete the wrong things
  - Restore will half-work
- Test: kill process before/after fsync, see which backup is valid

#3: Resume on checksums, not timestamps

- A backup dies halfway, you don't want to start over
- Tempting shortcut: compare `mtime` and skip unchanged files
- Postgres can change a file's contents within a single `mtime` tick
  - Now your backup is silently inconsistent
- SHA-256 the current file, compare → redo on mismatch
- Test: kill backup, flip bytes without `mtime` change, resume → check for redo

#6: fsync ordering on POSIX

- `fsync(file)` makes the bytes durable, but not the name
  - The name is an entry in the parent directory: separate data on disk
  - Creating or renaming a file changes the directory → `fsync(dir)` makes durable
- Every file close runs seven steps, in this order:
  - write tmp, fsync tmp, close tmp
  - open parent dir, fsync dir, rename tmp to final, fsync dir
- Bytes are durable before the filename is published
  → After a crash you find either tmp or final, never both
- Test: fault injection before/after each step

#6½: Atomic rename on every backend

- Every backend needs an atomic "become final" that cannot clobber
  - `posix`: `rename(2)` swaps the name in one step
    → `fsync(dir)` makes the swap durable
  - `sftp`: server-side `Rename`
- Object stores have no rename → copy if absent, then delete
  - `s3`: `CopyObject` with `If-None-Match: *`, then delete tmp
  - `azure`: `CopyFromURL` with `If-None-Match: *`
  - `gcs`: `Copier.If(DoesNotExist)`
  - Without this, invariant #2 would break on object stores

The rest of the rules

- #4: Retention cannot remove a backup a running backup needs
- #5: Probe the WAL archive before starting Postgres backup
- #7: Re-push a WAL segment: same bytes → no-op, different bytes → error
- #8: Backup from standby: check standby is actually replaying from primary
- #9: A backup has consistent encryption recipients, including workers
- #10: Multi-storage: backup complete when at least one backend commits

How do you test an invariant?

- We have to have some determinism in “kill it and see”
- Every durability step gets a named injection point
  - `StepWriteTemp`, `StepFsyncFile`, `StepRename`, ...
- Inject a failure at each step, then check what’s on disk
- Trust nothing: no status flags, no exit codes, no “works on my laptop”

Three design tenets

Tenet 1: Postgres never knows pgSafe exists

- No extension, no shared library, no hooks
- Cluster identity comes from SQL functions, not reading `pg_control`
  - `pg_control_system()`, `pg_control_checkpoint()`, …
- WAL archiving goes through `archive_command`, not `archive_library`
  - Costs 1 to 2 ms per 16 MB segment → practically nothing
- A backup tool must not be able to crash or block the database

Tenet 2: Standards and libraries first

- KISS, and “don’t reinvent the wheel”
- Postgres’s native `backup_manifest` format, not a custom one
  - `pg_verifybackup` can check our backups
- PG 17's WAL summarizer for incrementals
  - `pg_combinebackup` reconstructs the chain on restore
- Vendors' own SDKs for S3, Azure, GCS
- `age` for encryption, `net/http` and `crypto/tls` from the stdlib
- Less code our side → fewer bugs we have to own

Tenet 3: No credentials on the DB host

- The database host is the host most likely to be compromised
  - Probably not a good location for long-lived S3 keys
- So: the caller mints a narrow, short-lived credential per backup
  - S3: `AssumeRole` limited to `PutObject` on one prefix
  - Azure: service SAS, write and create only, HTTPS only
  - GCS: impersonated token, about an hour
- Delivered in memory over the worker channel.
  - Never on disk
- Exception: `archive_command` still needs a key on the DB host
  - But not for long

Three operating modes

PG-native or pgSafe mode?

PG-native (no worker)

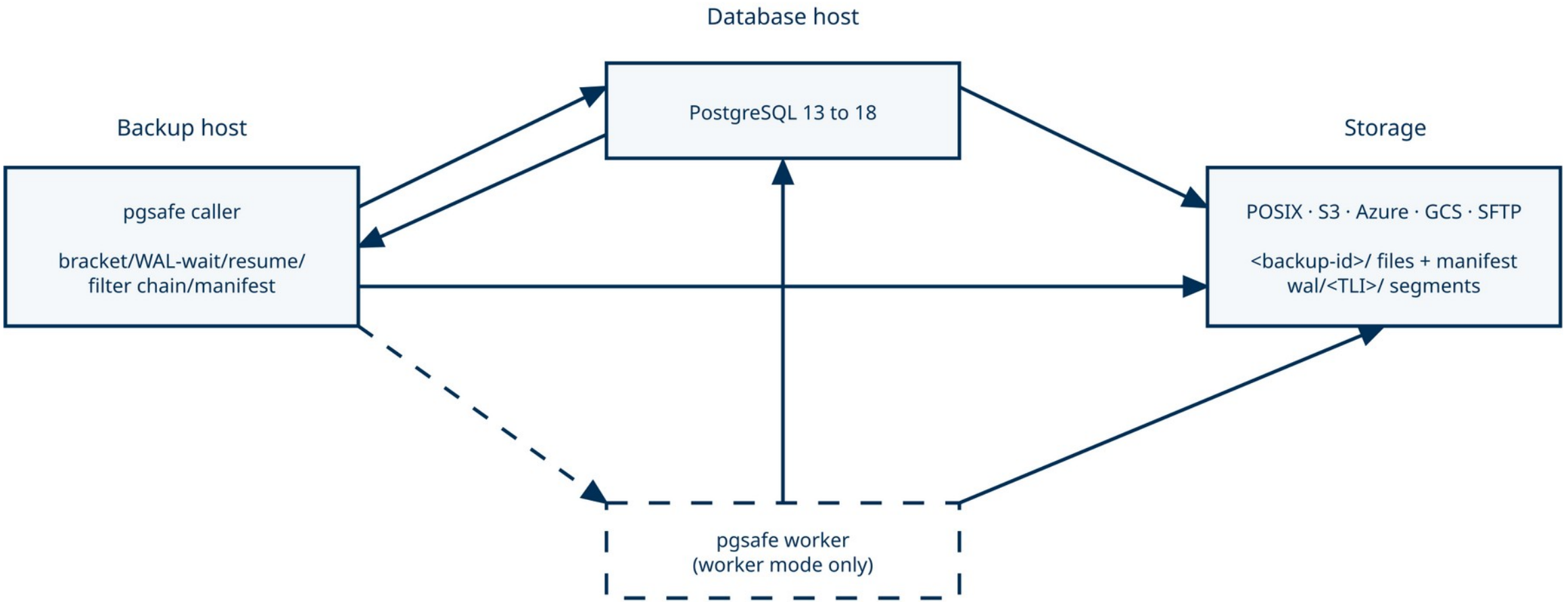
- `simple`: one replication connection, `pg_basebackup` tar stream
- `remote-parallel`: *n* connections calling `pg_read_binary_file()`
- Works wherever PG is connectable
- No shell access needed

pgSafe mode (worker)

- A worker process on the PG host reads `$PGDATA` via syscalls
- Same-host subprocess, or spawned over SSH
- Bytes move once:
  PG host → storage
- The only mode with scoped credentials

There is no `--mode` flag

- Mode is a property of your deployment, not via invocation
- Caller reads the config and works out the deployment shape
  - `pg.host` set and reachable over ssh → worker mode
  - Several workers, no `pg.host` → remote-parallel
  - Otherwise: simple
- `--workers` *n* is the only parallelism option to set
- Resolved topology is printed out in plain English


Database host
PostgreSQL 13 to 18
Backup host
pgsafe caller
bracket/WAL-wait/resume/
filter chain/manifest
Storage
POSIX · S3 · Azure · GCS · SFTP
<backup-id>/ files + manifest
wal/<TLI>/ segments
pgsafe worker
(worker mode only)

Where did 100,000 lines go?

Code accounting (i)

- More than 50% of C code had nothing to do with backups
- `common/type`: strings, lists, variants, JSON, pack → 12,833
- `common/io`: HTTP, TLS, sockets → 6,098
- `build/`: code generation for the config system → 6,034
- `config/parse.auto.c.inc`: generated option tables → 11,796
- S3, Azure, GCS, SFTP clients by hand → 8,353
- `protocol/`: worker coordination and wire format → 3,256
- Memory contexts, stack traces, error machinery → ~3,000

What's in those 51,000 lines?

- Things a C project in 2011 had to write itself:
  - HTTP client, TLS & socket layer
  - Request signing for S3, Azure and GCS by hand: SigV4, SharedKey, JWT
  - Wrappers around OpenSSL for ciphers and hashes, its own copy of MD5 (FIPS)
  - Strings, lists, variants, key-value maps, JSON, XML, a pack format
  - Memory contexts, `TRY/CATCH` macros
  - A code generator for the config system, and 11,796 generated LoC
- In Go: `net/http`, `crypto/tls`, `encoding/json`, generics and the GC
- Additionally, the cloud vendors' own SDKs for their APIs
- None of it is about backups – it's the cost of the language

Code accounting (ii)

- Several thousand more lines absorbed by Postgres itself
  - PG 17's WAL summarizer replaces a page-diff engine
  - `backup_manifest` replaces a private manifest format
  - `pg_combinebackup` replaces incremental reconstruction
  - `pg_verifybackup` replaces a verify engine
- `errgroup` replaces the worker pool, in a few dozen lines
- What's left?
  - The actual backup orchestration

Parallel file copy in Go

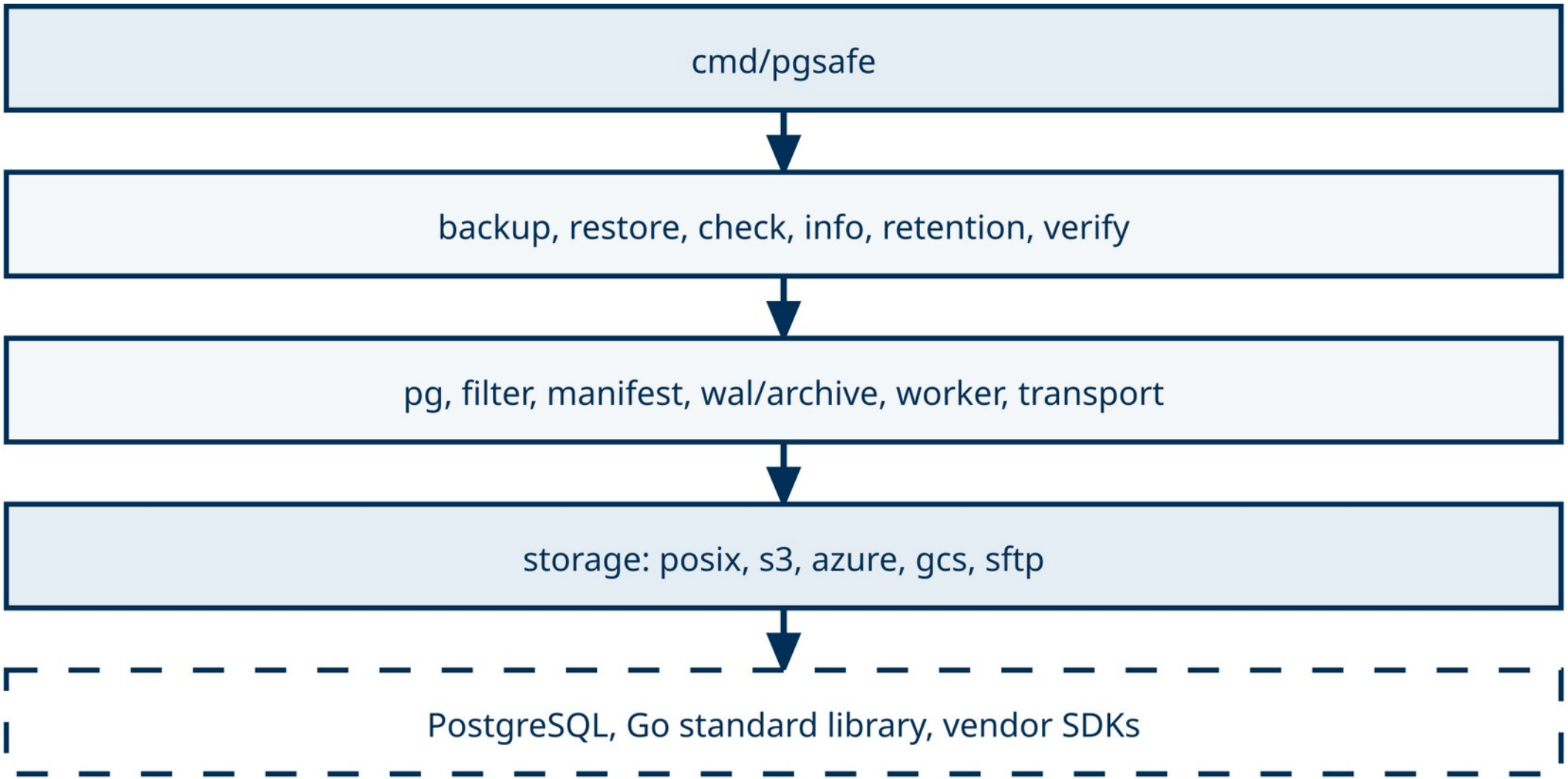
- pgBackRest `protocol/`: processes, handshakes, wire format → 3,256 lines
- pgSafe, parallel file copy:

```
g, ctx := errgroup.WithContext(ctx)
g.SetLimit(workers)
for _, f := range files {
  g.Go(func() error { return copyOne(ctx, f) })
}
return g.Wait()
```

- First error cancels the context: every worker sees it and stops
- Six lines, no supervision code, no wire protocol

The tally so far

- pgBackRest: **96,000** lines of C, plus 80,000 of tests
- pgSafe: **18,000** lines of Go, plus 15,000 of tests
- Five storage backends, six Postgres versions, three modes
- But this isn't about LoC
  - About how many people can understand the code & how easy it is to maintain
  - This is something one person could maintain in their spare time
- Caveats
  - pgSafe is much, much younger & does fewer things: `FUNCTIONALITY_GAPS.md`
  - Not tested against 10 years of user edge cases


cmd/pgsafe
backup, restore, check, info, retention, verify
pg, filter, manifest, wal/archive, worker, transport
storage: posix, s3, azure, gcs, sftp
PostgreSQL, Go standard library, vendor SDKs

The parts that cannot shrink

The resume protocol

- Every 10 files, checkpoint what arrived & hashes of stored bytes
- Next run lists the storage and decides (per directory)
  - Manifest present? → Completed, skip
  - No checkpoint file? → Nothing to resume
  - Checkpoint too old? → Delete stale backup attempt
  - Same version, type, parent backup, compression, keys? → Resume
- Then re-hash what is there, delete and re-upload mismatched
- `backup_label`/`tablespace_map` never reused, fresh ones for this attempt

The other three that cannot shrink

- WAL bracket guarantees
  - Ordering, timeline tracking, standby coordination, waiting
- Credential scoping
  - 4 backends, 4 different permissions systems
- Atomic rename substitutes
  - 5 backends, 5 different conditional write procedures
- Mostly contracts with things outside our process
  - Language choice can’t affect them

Testing (trust nothing)

- Unit, integration and end-to-end tests on every change, no exceptions
- Real PG13 to PG18 in containers for end-to-end
- Fault injection: deterministically crash writer at every step
- Local CI script is commit gate, CI runs the same steps: no divergence
- Backups that have never been restored are...?

  Not backups!

Re-architecting vs porting: worth it?

- Questions before you start
  - How much of the code is language baggage?
  - Has the platform absorbed some of it since it was written?
  - Can you write down the invariants without reading the code?
  - Is the test suite the real asset? You can port your understanding of it
- If some of these apply, a rewrite is smaller than it looks
- If none apply, you may be about to reproduce ten years of bugs
  - Or spawn new ones
- AI assistance saves you typing but can't produce the rules for you

Where we are & what is needed

- Open source, PostgreSQL licence
- Alpha code: read it, break it, tell me where the design is wrong
- `INVARIANTS.md` is an interesting file for contributors
  - Maybe there are more than 10
  - What have I not considered that can come back to bite us?
- Needed
  - More eyes for review 👀
  - Cloud backend testing
  - Feature gap filling

Recap

- Hard problems in a backup tool: ordering & durability, not moving data
- Write your invariants down, name them, and test them
- Standards and libraries first
- Keep credentials off the database host
- Half of an old C codebase can be the language rather than the problem
  - COUGH COUGH 🐘
- Test restore your backups

Thank you! Links:

- pgSafe repo: https://github.com/vyruss/pgSafe
- Socials:
  - YouTube: youtube.com/JimmyAngelakos
  - Mastodon: fosstodon.org/@vyruss
  - BlueSky: bsky.app/profile/vyruss.org
  - LinkedIn: linkedin.com/in/vyruss