PREPRINT · GW-2026-0003

Compiler-Hidden PostgreSQL Continuations for Direct-Style Server Programs

Readiness-driven libpq execution without ordinary worker occupation in the Gray runtime

Author: Annabella Proctor
ORCID: https://orcid.org/0009-0000-0799-1757
Affiliation: Grayworth
Publisher: Grayworth Technical Publications
Manuscript date: July 26, 2026
Version: 1.0
DOI: https://doi.org/10.5281/zenodo.21633225
ISBN-13: 9798191410890

Abstract

This paper presents Gray's compiler-hidden PostgreSQL continuation architecture, which preserves direct source-order database calls while moving connection and query waiting onto readiness-driven runtime state machines. The work responds to a retained one-hour baseline that delivered 3,999.85 requests per second at 4,000 requested but accumulated 802.82 ms corrected p99. The replacement path uses nonblocking libpq connection establishment, parameterized query dispatch, output flushing, input consumption, result draining, cancellation, and connection retirement under a normalized readiness reactor. A rate curve placed the new useful corrected-latency knee between 12,000 and 16,000 requested requests per second. A separate Static qualification with PostgreSQL 16, a 64-connection pool, four wrk2 threads, and 256 persistent HTTP connections delivered 7,961.48/s at 8,000 requested with 1.43 ms corrected p50, 4.60 ms p99, and 8.74 ms p99.9. During load, the runtime exposed as many as 164 simultaneously suspended handlers while using zero general task workers; after completion, active task, worker, and continuation counts returned to zero. Semantic gates covered overlapping queries with one configured task worker, typed errors after resumption, HTTP/2 multiplexing, cancellation and connection replacement, 404/500 behavior, and Adaptive deployment replacement. The result demonstrates that direct-style server code and readiness-driven database concurrency can coexist as one compiler-runtime contract without requiring application-authored polling logic or a worker per blocked handler.

Keywords: Gray; PostgreSQL; libpq; continuations; direct style; nonblocking I/O; readiness reactor

1. Introduction

Database access is where the promise of direct server code most often collides with the mechanics of evented execution. The application author wants to issue a query, receive a typed result, and continue in source order. The server runtime wants the thread that owns network progress to remain available while PostgreSQL accepts bytes, executes work, and returns data. Conventional systems expose that tension through callbacks, promises, explicit async syntax, fibers, coroutine libraries, or worker pools. Each is workable, but each asks either the programmer or a general scheduler to bridge two models.

Gray treats the bridge as a compiler-runtime responsibility. A registered handler may contain a synchronous-looking pg_query operation. When libpq cannot advance immediately, the compiler-produced program state is captured as a bounded handler continuation, the database operation is attached to socket readiness, and the HTTP reactor remains free to progress other connections. When the database state machine reaches a result or typed failure, the owning handler resumes at the source operation. The application does not allocate a task, maintain a pending-request map, or write a query-polling loop.

The work began from a clear failure. Gray's original route sustained 3,999.85 requests per second for an hour against 4,000 requested, yet corrected p99 reached 802.82 ms and p99.9 reached 1.55 seconds. The runtime was delivering work, but its waiting model generated an unacceptable tail. Replacing that path produced a rate curve whose useful knee moved between 12,000 and 16,000 requested requests per second, followed by a continuation-specific qualification at 8,000 requested with 4.60 ms p99. The transition is a major architectural result: it shows that the language's direct style need not imply blocking execution.

2. Research questions

The first question is semantic: can Gray preserve a direct, typed database call in handler source while compiling enough resumable state for suspension and later continuation? Success requires more than returning the common result. Locals, control position, typed catch behavior, response ownership, cancellation state, and source-located errors must remain valid across suspension. Live, Static, and Adaptive execution policies must agree even though their deployment lifecycles differ.

The second question is architectural: can connection establishment, output flushing, query waiting, and result collection proceed from socket readiness without occupying an ordinary Gray task worker? The strongest observable test is not a thread count alone. With GRAY_TASK_WORKERS fixed to one, two 250 ms PostgreSQL handlers must overlap. During HTTP load, the runtime must report suspended handlers while general task workers remain zero.

The third question is operational: does the design improve corrected latency and useful capacity relative to the blocking baseline while retaining bounded resources and complete cleanup? A rate curve is needed to locate the new knee. A focused qualification is needed to capture delivery, latency, suspended-handler concurrency, task-worker usage, response count, and final zero state. Cancellation must retire uncertain connections instead of returning potentially misaligned protocol state to the pool.

The fourth question is compositional. Independent HTTP/2 streams must suspend without serializing one another; missing routes and handlers that fail to respond must retain 404 and 500 behavior; and Adaptive replacement must permit an old in-flight continuation to finish while a replacement serves new requests. These behaviors determine whether continuations are an isolated benchmark mechanism or part of the server product.

3. Contributions

The primary contribution is a direct-style PostgreSQL execution model that uses compiler-hidden handler continuations rather than application-visible polling or one worker per waiting query. The runtime advances libpq through PQconnectStart and PQconnectPoll during connection setup, PQsendQueryParams for parameterized dispatch, PQflush for pending output, PQconsumeInput and PQisBusy for readiness-driven input, and repeated PQgetResult draining before a connection is considered reusable.

The second contribution is an explicit ownership and cancellation protocol. A suspended operation belongs to one handler continuation and one checked-out connection generation. Cancellation removes the operation from readiness tracking and discards the in-flight connection because wire-protocol alignment cannot be assumed. A regression then requires the pool to serve a subsequent query, proving that cancellation does not poison later borrowers. Completion, error, timeout, disconnect, and deployment replacement all converge on a bounded cleanup path.

The empirical contribution includes the original blocking baseline, a five-point 4,000-to-20,000 requested-rate curve, a separate 8,000/s readiness qualification, and the continuation-specific 8,000/s record. The final run delivered 99.519 percent of the offered rate with 4.60 ms corrected p99 while exposing 164 simultaneous suspended handlers and zero general task workers. Three hundred twenty-six successful stats observations and final zero counts provide direct runtime evidence for the concurrency claim.

The final contribution is a semantic gate matrix across execution policies, error paths, HTTP versions, and replacement behavior. This moves the claim beyond speed: Gray's continuation path preserves the language and server contracts that make direct syntax valuable in the first place.

4. Background and related work

PostgreSQL's libpq documentation distinguishes synchronous PQexec from the lower-level asynchronous interface. PQsendQueryParams dispatches a parameterized query without waiting. PQconsumeInput reads available server data; PQisBusy determines whether PQgetResult would block; PQflush advances queued output on a nonblocking connection; and PQgetResult must be drained until it returns null, including after errors. A select or poll loop can therefore coordinate many database sockets, but libpq intentionally leaves that coordination and application control flow to its caller.

Event-loop systems solve network scaling by keeping callback work short and returning control whenever an operation waits. Node.js documents this contract directly: one event loop orchestrates JavaScript callbacks and nonblocking I/O, while a worker pool handles selected expensive operations. Async/await improves source readability but still exposes suspension in the language. Thread-per-request and pool designs preserve direct style but consume schedulable workers and stack resources while operations wait. Go's goroutines offer a different strong solution by making lightweight concurrent execution a language/runtime primitive.

Continuations provide a general vocabulary for representing the remainder of a computation. Scheme's call-with-current-continuation demonstrates the expressive full form, while practical server runtimes often use narrower one-shot or delimited representations. Gray does not expose unrestricted first-class continuations. The compiler identifies supported suspension points in registered handlers and materializes only the state required to resume that handler once. This bounded form is easier to connect to request ownership, typed errors, and deployment lifetime.

The distinctive claim is therefore not the invention of nonblocking database I/O or continuations. It is their product integration. Gray combines direct handler syntax, compiler-known suspension, a readiness-driven libpq state machine, checked connection ownership, HTTP lifecycle behavior, and observable cleanup in one contract. The experiments test that complete path rather than timing a standalone asynchronous client.

5. Gray system architecture

A Gray HTTP handler runs under a request context owned by the reactor. The compiler emits resumable control metadata around database operations supported as suspension points. On entry to pg_query, the runtime acquires or queues for a pooled PostgreSQL connection. If query progress cannot complete immediately, it records the continuation, local state needed after the call, source identity, expected result type, cancellation token, and connection generation. The handler stops consuming execution capacity while its request remains live.

Database progress is managed by a dedicated normalized-readiness reactor. Connection creation is itself asynchronous: PQconnectStart yields a socket and PQconnectPoll advances through read/write requirements. Query dispatch uses PQsendQueryParams, then PQflush determines whether output remains. Read readiness triggers PQconsumeInput; PQisBusy protects against blocking result retrieval; available PGresult objects are processed and cleared until the null terminator establishes protocol completion. Pool exhaustion queues operations instead of converting temporary scarcity into immediate application failure.

The continuation resumes only after the runtime has converted the result or failure into a Gray value. Typed row decoding, binary parameters, transactions, and query errors therefore cross the same suspension boundary. A resumed typed catch sees the database error as if it had arisen at the direct call site. Source function identity remains attached for diagnostics. This is essential: a fast success-only path would not preserve the programming model.

Cancellation follows a conservative rule. Once a query is in flight, the runtime does not return that connection to the pool merely because the HTTP owner disappeared or requested cancellation. It removes readiness registration, requests cancellation where applicable, drains or closes as required, and retires uncertain state. Generation-checked ownership prevents a late readiness event from resuming a new borrower through a stale handle.

Adaptive replacement adds a second lifetime. The supervisor may install a new worker while an old handler is suspended. The old worker retains the continuation and resources needed to finish its accepted request; new traffic reaches the replacement. Draining completes only when old work is gone. The gate therefore tests continuity across application replacement, not just within a steady process.

6. Experimental design

The historical baseline is the PostgreSQL phase of the one-hour route matrix. It used Gray Static, PostgreSQL on declared local CPUs, 256 persistent HTTP connections, four wrk2 threads, and 4,000 requested requests per second. It delivered 3,999.85/s but accumulated 802.82 ms corrected p99. This result establishes the problem but is not treated as a controlled before/after measurement against the final continuation fixture because runtime revision, fixture details, and host state differ.

The intermediate readiness curve used one HTTP reactor process, one PostgreSQL readiness thread, GRAY_TASK_WORKERS=1, 256 connections, four wrk2 threads, a three-second warmup, and fifteen seconds per point. Requested rates were 4,000, 8,000, 12,000, 16,000, and 20,000/s. Gray used CPUs 2-3, wrk2 used CPUs 4-7, and the controller used CPU 0. The curve's purpose was to expose the transition from stable delivery to queue growth rather than to choose the highest achieved count.

A separate 30-second readiness qualification at 8,000 requested requests per second required at least 95 percent delivery, bounded corrected latency, zero sampled RSS growth, zero descriptor growth, and successful exact responses. The later continuation-specific qualification also used PostgreSQL 16, a pool of 64, four wrk2 threads, 256 persistent connections, and 8,000 requested/s for 30 seconds. Runtime stats were sampled independently during load.

Semantic tests were separated from the throughput run. Two 200 ms handlers and two 200 or 250 ms PostgreSQL handlers had to overlap. Database failures had to reach typed catches after resumption. HTTP/2 multiplexed handlers had to suspend independently. Cancellation had to produce the declared error and leave the pool usable. Missing routes, missing responses, and Adaptive old/new request behavior had explicit expected outcomes. Sanitizer configurations exercised the same ownership paths.

The central measured claim is based on the continuation qualification itself: 7,961.48/s achieved at 8,000 requested, 4.60 ms corrected p99, 164 maximum suspended handlers, and zero maximum general task workers. The larger improvement relative to the old baseline is architecturally consistent and highly consequential, but this paper does not compute a causal speedup from two non-identical runs.

7. Reproducibility identity and controls

Grayworth's performance records use an evidence-first publication rule. A headline value is admitted only after the runner has retained the command contract, executable identity, fixture identity, environment description, raw measurements, and a machine-readable completion marker. The report generator consumes verifier-accepted evidence rather than terminal excerpts or manually selected screenshots. This rule matters because a benchmark is not merely a timer around a program: it is a claim about which program ran, under what resource limits, against which request stream, and with what correctness checks. Preserving those facts makes later reanalysis possible even while Gray itself continues to change.

The HTTP experiments use an open-loop, constant-throughput load model through wrk2. In a closed-loop client, a slow response delays the next request and can make an overloaded service appear to have acceptable latency because the client stops offering work at the intended schedule. wrk2 records latency from the time a request should have been sent, exposing that coordinated-omission effect. Requested rate, achieved rate, delivery ratio, and corrected latency are therefore reported together. Throughput without latency can reward an ever-growing queue; latency without achieved delivery can describe a server that simply did not process the offered work.

Every HTTP route is preflighted for exact status, body, and content type before load begins. Warmup is separated from the accepted interval, and the server process is monitored throughout the interval. A point fails on socket errors, non-success status codes, unexpected process exit, changed fixture identity, or missing completion evidence. The one-hour matrix additionally samples resident memory and open descriptors after a settling window. Flat sampled values do not prove the absence of every leak, but they do show that the accepted workload did not produce monotonic resource growth at the recorded sampling resolution.

CPU placement is treated as part of the experiment rather than an incidental host detail. Server, load generator, controller, and database roles use declared CPU sets where the retained runner supports them. The same-host comparison constrains every server fixture to one CPU, 512 MiB of memory, and 256 processes inside one digest-pinned Ubuntu image; wrk2 runs outside that container on disjoint client CPUs. This isolates the application-facing runtime path while holding the operating-system image, cgroup contract, network path, and load generator constant.

A controlled-delivery point in the comparison is defined before interpretation as at least 95 percent of the requested rate with corrected p99 below 100 ms. It is a reporting heuristic for locating a useful throughput knee, not a universal service-level objective. The complete curve remains primary evidence because a single threshold can hide how abruptly queues form after saturation. The papers therefore distinguish the highest point satisfying that contract from the maximum achieved throughput and from the later one-hour operating point selected below the knee.

Fresh-process and interleaved sampling are used where process startup, dynamic compilation, or thermal drift could otherwise bias the result. Output is checked before a timing sample is accepted. Medians are reported for repeated compute diagnostics because they resist a single scheduling interruption, while minimum and maximum values show the observed spread. HTTP curves retain every rate point rather than reporting only the favorable region. Negative results, including the original blocking PostgreSQL tail latency and Gray's own collapse at 75,000 requested requests per second, remain in the record because they identify the architectural boundary that the next experiment must address.

The publication language separates observation from interpretation. An observed value is copied from retained output; an architectural explanation is supported by implementation structure, profiling, or an intervention; a projection is labeled as future work. Grayworth publication identifiers remain stable internal record keys, while each version 1.0 preprint also carries its reserved DOI for the archival record. A later artifact release may add source bundles without changing these canonical record URLs, DOI identities, or original measurements.

Reproducibility does not require disclosing unrelated commercial implementation details. It does require enough identity to tell whether two records describe the same executable and fixture. For that reason, the appendices retain cryptographic hashes, runtime modes, requested rates, duration, connection counts, thread counts, database versions, and named artifact directories. Public benchmark and report files can be released progressive-

ly toward Gray v1. Until then, the paper distinguishes currently downloadable manuscript and metadata files from repository artifacts named for later release. No unavailable artifact is described as independently archived or externally replicated.

Table 1. Continuation qualification identity

Property	Retained value
Runtime SHA-256	b197c99e082769ce55d884dffae0754df1e28541610718dd29b4400087e2a790
Fixture SHA-256	d6f0155ad91ab2282ac77a347132b68eb715cf7c7be44099382eeca827395069
Database	PostgreSQL 16, isolated local container
Pool	64 connections
Load	wrk2 -t4 -c256 -d30s -R8000 --latency
Mode	Static

8. Results

The readiness curve delivered 3,965.82/s at 4,000 requested with 4.53 ms corrected p99; 7,809.53/s at 8,000 with 21.66 ms p99; and 11,879.17/s at 12,000 with 49.41 ms p99. At 16,000 requested, delivery fell to 94.666 percent and p99 rose to 805.89 ms. At 20,000, delivery fell to 75.263 percent and p99 reached 3.40 seconds. The useful knee therefore lies between 12,000 and 16,000 requested requests per second under that fixture.

The separate 8,000/s readiness qualification achieved 7,904.47/s, or 98.806 percent delivery, with 2.82 ms p50, 32.80 ms p99, and 54.65 ms p99.9. Sampled RSS was 14,928 KiB with zero growth, and descriptors remained at 327. These values established that the explicit readiness state machines worked under sustained load before compiler-hidden continuation integration was evaluated.

The continuation qualification achieved 7,961.48/s at 8,000 requested, or 99.519 percent delivery. Corrected p50 was 1.43 ms, p99 4.60 ms, p99.9 8.74 ms, and the observed maximum 25.86 ms. wrk2 received 238,852 responses in 30 seconds. The sampler completed 326 successful stats observations and observed as many as 164 simultaneously suspended HTTP handlers.

General task workers remained zero throughout the continuation qualification. After load, active_tasks, general_task_workers, and http_continuations were all zero. The result directly demonstrates that the suspended handlers were represented as reactor-

owned continuation state rather than workers parked on libpq calls. Because 164 suspended handlers coexisted while no general worker existed, the evidence is stronger than a configuration assertion.

Every semantic gate passed: overlapping ordinary and PostgreSQL handlers, typed query failures, independent HTTP/2 streams, cancellation and subsequent pool health, 404 for missing routes, 500 for handlers that produce no response, and Adaptive replacement with an old in-flight request completing while the new worker served new traffic. Live and Static paths agreed on expected behavior.

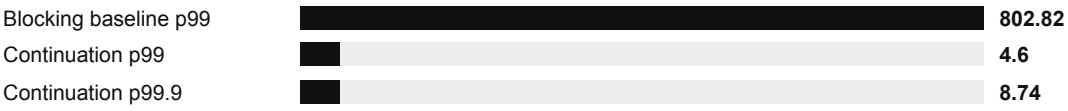
Table 2. Readiness-driven saturation curve

Requested	Achieved	Delivery	p50	p99	p99.9
4,000/s	3,965.82/s	99.146%	2.21 ms	4.53 ms	6.50 ms
8,000/s	7,809.53/s	97.619%	2.63 ms	21.66 ms	40.51 ms
12,000/s	11,879.17/s	98.993%	3.50 ms	49.41 ms	58.72 ms
16,000/s	15,146.64/s	94.666%	550.40 ms	805.89 ms	849.41 ms
20,000/s	15,052.65/s	75.263%	2.30 s	3.40 s	3.48 s

Table 3. Final continuation qualification

Requested	Achieved	p50	p99	p99.9	Suspended	Workers
8,000/s	7,961.48/s	1.43 ms	4.60 ms	8.74 ms	164 max	0 max

Figure 1. Normalized reported measurements



9. Analysis and interpretation

The experiment answers the architectural question directly. Direct-looking source did not force thread-blocking execution. The compiler and runtime transformed the database boundary into a one-shot continuation, and the stats record showed that a large population of handlers could wait without consuming the ordinary worker pool. This is precisely the coexistence Gray set out to establish: sequential source reasoning for application authors and readiness-driven multiplexing for the runtime.

The corrected-latency result is equally important. The old route could deliver 4,000/s while allowing nearly a second of p99 queue delay. The continuation run delivered almost 8,000/s with 4.60 ms p99. Because the runs are not a matched A/B, the paper does not attribute a formal 175-fold latency improvement to one code change. It can conclude more strongly where the evidence is direct: the shipped continuation architecture operated at the final recorded rate with a tight tail and without general worker occupation, resolving the failure mode that motivated it.

The 12,000-to-16,000 curve transition shows that removing worker blockage does not remove finite capacity. Database execution, pool availability, reactor work, response processing, and CPU remain bounded resources. At 16,000, queueing becomes visible before raw achieved throughput stops increasing. This is why corrected latency and delivery must be read together and why the selected 8,000 qualification point is an operating record rather than an attempted peak.

The semantic results distinguish this mechanism from a benchmark-only callback. Typed errors survive resumption, streams remain independent, cancellation protects pool state, and deployment replacement respects old work. The continuation is integrated with Gray's ownership model. That integration is the scientific and product significance of the work: concurrency is not merely hidden syntax; it is a lifecycle contract observable at failure boundaries.

10. Engineering implications

Application authors can write the database sequence they mean without manually splitting control flow around readiness events. This reduces the surface for mismatched pending maps, forgotten cleanup, unhandled late callbacks, and source locations detached from failures. The compiler has enough knowledge to preserve only the state needed after the supported suspension point, while the runtime has enough ownership information to cancel and retire resources safely.

Runtime capacity becomes less coupled to ordinary worker count. General workers remain available for genuinely blocking foreign calls or CPU work instead of being sized for the number of database waits. Operators gain more meaningful observability: suspended continuation count reflects waiting application operations, while task-worker count reflects a different resource. The final-zero assertion turns that distinction into a leak gate.

The conservative cancellation rule is operationally important. Returning an uncertain connection to a pool can turn one canceled request into a later request's protocol corruption. Discarding or completely draining that connection costs capacity in the cancellation case but protects correctness. Generation checks similarly trade a small amount of bookkeeping for immunity to stale readiness delivery.

The design opens further work without making it a prerequisite for the result. Prepared statements, pipeline mode, row streaming, backpressure across large result sets, richer transaction scopes, and distributed database latency can reuse the same continuation and ownership model. Each should be measured separately because it changes the database-side workload, but the difficult control-flow bridge has already been demonstrated.

11. Threats to validity

The experiments use a local PostgreSQL 16 container and a small parameterized query. Network distance, storage pressure, locks, query plans, schema shape, result size, transaction contention, and pool sizing can change latency and capacity substantially. The result demonstrates the runtime's waiting architecture, not a throughput guarantee for arbitrary production databases. The local setup is useful because it makes runtime overhead and queue formation visible without wide-area variance.

The old blocking baseline and final continuation run differ in runtime identity and fixture history. Their contrast motivates and explains the work but is not a randomized controlled A/B. The central conclusion therefore rests on direct final-run observations: corrected latency, achieved delivery, suspended-handler count, zero worker count, and semantic gates. A future release should rerun matched blocking and continuation fixtures from the same commit if a numerical causal speedup is desired.

Stats sampling is observational and finite. A maximum of 164 suspended handlers means at least that many were observed, not that the true instantaneous maximum could not have been higher between samples. Zero observed general workers is reinforced by configuration and the overlap gate, but operating-system threads used for the HTTP and PostgreSQL reactors still exist; the claim concerns Gray's general task-worker pool, not literal zero threads.

Thirty-second qualification and fifteen-second curve points do not establish day-scale resource stability. Sanitizer and cleanup gates reduce memory-safety and ownership risk but cannot prove the absence of all defects. Commercial runtime source is not public in

version 1.0, so independent source builds await the Gray v1 artifact release. The paper provides exact binary and fixture hashes to preserve identity until then.

12. Artifact availability

The canonical manuscript, PDF, citation formats, metadata preview, and stable publication record are available at GW-2026-0003. Retained internal evidence is named by benchmarks/http/db-continuation-report.md, benchmarks/http/db-readiness-report.md, db-readiness-single-reactor-15s-v2, and db-readiness-qualification-30s-8k. The public engineering and research pages describe the surrounding runtime architecture.

The planned Gray v1 bundle will include releasable fixtures, runner commands, raw wrk2 output, stats samples, semantic test descriptions, and build identities. The version 1.0 preprint uses DOI 10.5281/zenodo.21633225; future artifact additions will keep the publication ID, DOI, canonical URL, author, manuscript date, and reported numbers stable.

13. Conclusion

Gray demonstrated that direct-style PostgreSQL handler code can execute through compiler-hidden continuations and readiness-driven libpq state machines without occupying ordinary application workers. At 8,000 requested requests per second, the final Static qualification delivered 7,961.48/s with 4.60 ms corrected p99 and 8.74 ms p99.9. The runtime exposed 164 simultaneous suspended handlers, zero general task workers, and complete return to zero after load.

The result resolves the architecture exposed by the earlier 802.82 ms p99 baseline. It does so without making application authors write callbacks, pending maps, or polling loops, and without discarding typed errors, cancellation, HTTP/2 independence, or Adaptive replacement semantics. The finding is larger than a database benchmark: it establishes a compiler-runtime technique for reconciling direct programs with evented systems behavior, implemented and measured as part of a working server language.

14. References

1. Gil Tene. wrk2: a constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2
2. PostgreSQL Global Development Group. Asynchronous Command Processing in libpq. https://www.postgresql.org/docs/current/libpq-async.html
3. The Go Authors. Package net/http. https://pkg.go.dev/net/http

4. The Go Authors. Diagnostics. https://go.dev/doc/diagnostics
5. Node.js Project. Don't Block the Event Loop or the Worker Pool. https://nodejs.org/en/learn/asynchronous-work/dont-block-the-event-loop
6. PHP Documentation Group. FastCGI Process Manager configuration. https://www.php.net/manual/en/install.fpm.configuration.php
7. M. Anton Ertl and David Gregg. The Structure and Performance of Efficient Interpreters. https://jilp.org/vol5/v5paper12.pdf
8. LLVM Project. Garbage Collection Safepoints in LLVM. https://llvm.org/docs/Statepoints.html
9. Revised6 Report on the Algorithmic Language Scheme. https://r6rs.org/final/html/r6rs/r6rs.html
10. Grayworth. Gray comparison methodology and retained dataset. https://grayworth.com/compare/methodology
11. Grayworth. Gray performance research record. https://grayworth.com/research
12. Grayworth. Formal HTTP comparison data. https://grayworth.com/data/http-comparison-results.json

Appendices

Appendix A

Appendix A records the final identities. Runtime SHA-256: b197c99e082769ce55d884dffae0754df1e28541610718dd29b4400087e2a790. Fixture SHA-256: d6f0155ad91ab2282ac77a347132b68eb715cf7c7be44099382eeca827395069. The command contract was wrk2 with four threads, 256 connections, 30 seconds, 8,000 requested requests per second, corrected latency, and the local /db route.

Appendix B

Appendix B records semantic acceptance: ordinary overlap; PostgreSQL overlap with one configured worker; typed query error after resumption; binary parameters and typed rows; transaction behavior; injection-resistant parameterization; cancellation followed by successful pool reuse; independent HTTP/2 stream suspension; missing-route 404; missing-response 500; Adaptive old-request completion and new-request service; final zero tasks and continuations.

Appendix C

Appendix C defines terminology. A handler continuation is the bounded runtime representation of a registered handler's remaining computation at a supported suspension point. A general task worker is a member of Gray's ordinary worker pool. A readiness reactor is the component that advances nonblocking socket state after read/write readiness. These categories are distinct, so the paper does not use zero general workers to imply zero runtime threads.