Universal Library of Engineering Technology Book

ISSN: 3064-996X | Volume 3, Issue 2

Open Access | PP: 60-71

DOI: https://doi.org/10.70315/uloap.ulete.2026.0302011

Methodology for the Experimental Evaluation of Memory Allocation Strategies and Object Lifetime Models in C++ Systems

Maksim Martynov

Abstract

The methodology considers the experimental evaluation of memory allocation strategies and object lifetime models in C++ systems as a reproducible research process grounded in a generative workload model, a phase-based execution structure, and an event-driven measurement scheme. The methodology's relevance stems from the growing role of allocators in the performance of computing systems, where differences in memory placement and reclamation policies affect latency, fragmentation, peak resource consumption, and resilience under prolonged load. The aim of the work is to formalize a procedure for allocator comparison by elevating object lifetime to the status of a fundamental experimental parameter. The novelty of the approach lies in the integration of deterministic operation generation, explicit lifetime models, the RampUp, Steady, and BulkReclaim execution phases with churn modeled as a workload profile applied within them rather than as a separate structural phase, as well as a comparability protocol fixing the workload, seed, environment, the rules for accounting for non-applicable operations, and a capability-gated validation matrix that excludes invalid allocator × workload × lifetime combinations. The principal conclusions are that accounting for lifetime reveals the mechanisms of memory degradation arising from the intermingling of short-lived and long-lived objects. Configuring the methodology with arena separation by lifetime profile reduces the measured memory footprint overhead and stabilizes the live and peak reserved-byte counters tracked by the framework. Phase decomposition exposes the differences between warm-up, steady-state mode, churn load, and bulk memory return. The methodology will be useful for researchers, infrastructure software developers, performance engineers, and architects of C++ systems.

Keywords: *C++, Memory Management, Allocators, Object Lifetime, Memory Fragmentation, Experiment Reproducibility.*

INTRODUCTION

Contemporary C++ development is defined by the conjunction of high-performance requirements and increasingly complex resource management in multicore and heterogeneous environments (Cuadros Zegarra et al., 2024). Dynamic memory management remains one of the most consequential yet least formalized areas of systems programming (Bailleu et al., 2024). Moreover, allocator selection is still frequently based on outdated metrics or subjective preferences, even though it can affect overall system efficiency (Bell et al., 2024; Li et al., 2025). While these contexts are inherently concurrent, the present methodology isolates baseline allocator behavior in a single-threaded setting; multithreaded and NUMA-aware extensions are addressed in companion publications of the same research line.

The memory wall problem necessitates a reconsideration of allocator evaluation methods. While tests like the Larson test and Google Threadtest, which both focus on throughput under multithreaded contention (Li et al., 2025), are commonly used to measure native allocator throughput, they do not consider object lifetime semantics and how these impact long-term heap fragmentation (Maas et al., 2024). However, workloads such as game engines, client/server workloads, and low-latency trading systems proceed by allocating and deallocating the objects in a cyclic pattern as per frames, sessions, and transactions (Zhou et al., 2024). The Larson test and Google Threadtest are referenced here as background examples that motivate the present work. They are not part of the proposed methodology's experimental matrix.

This work formalizes the experimental evaluation of memory allocation strategies by treating object lifetime as the primary experimental parameter. The interaction of allocator performance with memory-safety mechanisms, pointer nullification, guard zones, delayed reuse, is intentionally treated as out of scope here and is addressed in companion publications of the same research line (Nam, 2024; Hong et al., 2025). Likewise, while game engines, client/server workloads, and low-latency trading systems motivate the

Citation: Maksim Martynov, "Methodology for the Experimental Evaluation of Memory Allocation Strategies and Object Lifetime Models in C++ Systems", Universal Library of Engineering Technology, 2026; 3(2): 60-71. DOI: https://doi.org/10.70315/uloap.ulete.2026.0302011.

present work, the methodology abstracts their behavior into parameterized, reproducible workload generation models without assuming any specific application.

The novelty of the proposed methodology rests on the joint operation of five interlocking design choices, none of which appears as a coordinated set in earlier allocator evaluation schemes. Object lifetime is elevated to a primary experimental variable through the function $L:A\to\mathbb{N}$ and a closed family of policies covering FIFO, LIFO, Random, Bounded, and LongLived regimes, whereas throughput-oriented benchmarks of the Larson and Threadtest lineage leave the temporal dimension of the live set outside their parameter space and concentrate on multithreaded contention. The deterministic workload model $W=(O, P, S)$ establishes a reproducibility invariant under which a fixed pair of parameters and generator seed yields an identical operation stream across runs, a guarantee absent from trace-driven approaches once the captured scenario is altered or extended.

Phase decomposition into RampUp, Steady, and BulkReclaim, with churn realised as a workload profile applied within Steady rather than as a structural phase of its own, separates the cost of allocator data structure initialization, the behavior of the steady-state regime, the effects of intensive object rotation, and the quality of bulk page return to the operating system, which removes the averaging of heterogeneous regimes into a single aggregate figure. The measurement subsystem is formulated as a function $M=f(E)$ over an event stream with tick-based snapshots anchored to a configurable operation count, which permits reconstruction of degradation trajectories and decouples data acquisition from workload execution.

The comparability contract enforces identical workloads, seeds, phase schemes, and environments, while the rule for separate treatment of non-applicable operations through the N/A marker, together with the capability-gated matrix of admissible allocator × workload × lifetime combinations, eliminates a class of comparison errors in which zero cost attributed to an absent operation masks an architectural distinction between implementations. The conjunction of these five elements is absent from previously published evaluation frameworks. Trace-based schemes of the Wilson type record program behavior without lifetime parameterization, synthetic stress tests of the Larson family omit the phase structure of memory exploitation, and fragmentation analyses lacking an event-driven measurement model cannot bind peak consumption and overhead ratios to a specific phase and lifetime policy.

CHAPTER 1. FORMAL MODEL OF EXPERIMENTAL EVALUATION

To construct a scientifically substantiated methodology, it is necessary to move away from empirical testing and toward a rigorous formalization of the experimental conditions. This chapter describes the mathematical and conceptual foundations that enable the representation of workload, lifetime, and the measurement process as a single integrated system.

Workload Model

Within this methodology, workload is defined as a deterministic generative process. This permits flexible configuration of experimental parameters while preserving the ability to fully reproduce the conditions. Formally, workload W is represented as a tuple of three components: $W=(O, P, S)$, where O is the stream of operations, P is the parameter space, and S is the seed of the random number generator. The operation stream O is a sequence of tuples $op=(type, size, alignment, metadata)$, where the type of operation is $type \in \{alloc, free\}$, and size and alignment determine the physical requirements of the requested block. The size and alignment fields carry physical meaning only for alloc operations; for free, they are typically zeroed or ignored.

Implementation note: The tuple $W = (O, P, S)$ is a conceptual framework. In the reference implementation, parameters P and seed S are encapsulated in a WorkloadParams configuration object, while the operation stream O is not materialized in memory but produced lazily by a stateful, deterministic OperationStream generator. The operation tuple $op = (type, size, alignment, metadata)$ similarly abstracts a richer runtime structure that additionally carries the operational fields reason, tag, flags, and ptr.

The parametric space P includes the statistical distributions that determine the character of the workload. One key characteristic is the distribution of object sizes. The table below systematizes the principal types of distributions used to model various classes of C++ systems.

Table 1. Main types of distributions used to model various classes of C++ systems

Distribution Type	Characteristic	Application Area
Uniform	Evenly distributed within the range [min, max]	Synthetic evaluation of data structure stability
Normal	Most values concentrated around the mean size	Modeling homogeneous objects, for example graphics elements
Pareto	Heavy tail, with many small and few large values	Realistic modeling of server workloads and databases

Bimodal	Two distinct size peaks	Scenarios with separation into metadata and data buffers
LogNormal	Skewed distribution	Typical for many system calls in POSIX environments
Custom Buckets	User-defined intervals	Precise reproduction of the memory profile of specific software

Of particular importance in the model is the determinism parameter. For any identical input data P and S, the generation function G(P, S) must produce an identical sequence of operations O. This condition constitutes an invariant of the methodology, ensuring direct allocator comparison. If the sequences of operations differ by even a single position, the comparison of metrics becomes invalid due to the altered heap state.

Implementation note: The size distributions in Table 1 illustrate one axis of the parameter space P. The full WorkloadParams structure additionally exposes the alloc/free ratio, lifetime model selector, maximum live-set bound, alignment distribution, total operation count, and tick interval. Each of these is a first-class configuration parameter. Strict reproducibility additionally depends on the dynamic execution state, for example, the current liveCount tracked by the LifetimeTracker, being identical across runs. In practice this is satisfied automatically when (P, S) and the phase configuration are held constant, but the dependency should be acknowledged.

For practical application of this model, first fix the system class and select the object size distribution that most closely approximates the actual operational picture. For a graphics engine, it is reasonable to use a normal distribution centered around the typical size of a scene element; for a server-side request-processing module, a Pareto distribution; for a system with small service objects and large buffers, a bimodal distribution. Then set the same initial generator value and the same set of workload parameters, and run several memory management strategies through an identical sequence of allocations and deallocations.

For example, if in a network service about 80 percent of objects have a size of 64–256 bytes, while the remaining 20 percent occupy 4–32 kilobytes, a workload with such proportions should be formed, the generator value should be preserved, a series of one million operations should be executed, and allocation time, the number of calls to the system allocator, and the memory footprint proxies (overhead ratio of reserved to live bytes, peak reserved bytes) should be compared. Such an order enables the acquisition of a reproducible behavioral profile and the selection of the strategy best suited to the specific operational environment.

Object Lifetime Model

Object lifetime has traditionally been ignored in throughput-oriented benchmarks, leading to a failure to understand the causes of memory fragmentation. In the proposed methodology, lifetime is introduced as a first-class parameter, defined by the function $L:A\to\mathbb{N}$, where A is the set of completed memory allocations, and the value of the function corresponds to the ordinal number of the operation O in the stream at which the object must be freed. This allows modeling when an object dies and the order in which this occurs relative to other objects.

Implementation note: The function $L: A \to N$ is the formal model. The reference implementation realises it through a LifetimeTracker that stores per-allocation metadata, notably the allocation index allocTime, and applies a policy-driven selection at each free operation. The result is functionally equivalent for FIFO, LIFO, Random, Bounded, and LongLived policies, but does not require pre-computing or materialising the full L mapping in advance.

Lifetime policies determine the dynamics of the set of live objects. From there, the methodology identifies several key models. For example, the FIFO (First-In-First-Out) model corresponds to queues for processing messages, while the LIFO (Last-In-First-Out) model is typical for stack allocators and nested contexts of execution. LIFO is also typical in functional C++ code. The Random model analyzes the performance of an allocator under random fragmentation. The Bounded model imposes a condition on the cardinality of the set $|\text{Live}(t)| \leq C_{\text{live}}$, forcing the system to free old objects when the capacity limit is exceeded. The LongLived model represents objects that survive most allocation cycles, creating islands of occupied memory that impede OS page defragmentation.

It is precisely the combination of different lifetime models that enables the detection of allocator degradation. For example, mixing objects with FIFO and LongLived policies within a single memory region leads to external fragmentation, since long-lived objects lock pages and prevent the allocator from returning them to the operating system. The methodology supports the proposition that fragmentation results from the interaction between placement strategy and lifetime patterns.

Implementation note: In the current framework, mixed-lifetime effects are studied indirectly by comparing isolated single-policy scenarios (for example, malloc_fifo_churn vs. malloc_longlived_churn). A unified execution mode that runs multiple lifetime models simultaneously within one workload is on the roadmap (see §4.2).

To avoid external fragmentation when mixing objects with different lifetimes, separate memory arenas should be used for each lifetime model. The lifetime profile of an object should be determined at the stage of its creation, and allocation should then be directed to the corresponding region. Short-lived messages following the FIFO model should be placed in a fixed-size ring buffer, while long-lived caches should be moved to a separate arena with its

own allocator. The framework's memory footprint proxies, peak reserved bytes, live bytes, overhead ratio, should be measured under a stable load, and fragmentation should be tracked through the overhead ratio of reserved-to-live bytes exposed by the LifetimeTracker rather than via direct OS-level introspection.

For instance, consider an illustrative scenario in which a C++ trading gateway is being developed that receives 50000 market quotes per second. The figures that follow are illustrative of the qualitative effect; direct OS-level RSS instrumentation lies outside the scope of the current framework, which surfaces the same trends through reserved-byte and overhead-ratio proxies. Each quote lives for about 2 milliseconds before being passed to a strategy, while the system also holds a long-lived instrument directory containing 200000 entries and a pool of network buffers. When a shared allocator via standard new is used, the process's RSS grows from 800 MB to 2.4 GB after an hour of operation, even though the amount of actually live data is always about 600 MB. The solution is as follows. Arena A1 of 128 MB should be allocated for quotes with a ring allocator, where the write pointer moves in a circle and old objects are overwritten according to the FIFO model. Arena A2 of 64 MB should be allocated for the instrument directory with a bump allocator that does not free memory at all until process termination. Arena A3 of 32 MB should be allocated for network buffers with a pool allocator on 4 KB blocks. operator new should be overloaded for the Quote, Instrument, and Buffer classes, respectively. After such separation, RSS stabilizes at 230 MB and stops growing, because long-lived instruments no longer lock pages containing long-dead quotes.

Phase Model of Execution

The process of experimental investigation in the methodology is represented as a composition of sequential phases E={ ϕ_1, ϕ_2, ..., ϕ_n} where each phase has its own specific task and workload configuration. Decomposing into phases allows isolating different effects, such as the initialization cost of allocator data structures or the efficiency of OS memory-return mechanisms.

A typical experiment consists of four key stages. The first is the RampUp phase, during which the system is warmed up, and the heap is initially filled until the target live-set volume is reached. The second is the Steady phase, characterized by a stable load in which the number of allocations equals the number of deallocations. This enables measuring performance in the steady-state regime. The third is a Churn workload, representing intensive object rotation under constant or pulsating memory volume, applied within a Steady-class phase rather than as a structural phase of its own. It is here that problems with free-block search and merge efficiency become manifest. The fourth stage is the BulkReclaim phase, imitating the mass destruction of objects, for example, during a game scene switch or request completion.

Implementation note: In the reference implementation, RampUp, Steady, and BulkReclaim are structural phase types in the execution pipeline, while Churn is a workload profile applied within a Steady phase rather than a phase of its own. The canonical pipeline is therefore RampUp → Steady (optionally with churn-class workload) → BulkReclaim, with Drain and Evolution as registered variants of BulkReclaim and Steady respectively.

The methodology also supports the semantics of open-ended execution, where the number of operations is not specified in advance. In this mode, execution of the phase continues until an external event occurs or a termination predicate is satisfied. This is critical for testing real-time systems, where response time over an arbitrary interval is more important. The framework supports user-defined termination predicates of arbitrary form; thresholds based on p99 latency or reserved-byte ceilings are not built-in primitives, but can be expressed as custom predicates in scenario configuration.

For the phase model to yield useful data, the experiment should be designed so that each phase answers a separate question about allocator behavior, and their measurements should not be conflated into a single averaged figure. Before launch, the target live-set size and the exit criterion for each phase should be determined. Between phases, a snapshot of heap metrics should be taken, and latency counters should be reset. Otherwise, the tails of the distribution from RampUp will distort the picture of Steady. The Churn phase should be launched only after Steady has shown a stable median latency for at least two minutes. For open-ended execution, the termination predicate should be defined through the p99 latency threshold or RSS exceedance, so that the test stops upon degradation rather than by a fixed timer.

Implementation note: The stability gates and per-phase orchestration described above, for example, requiring a stable median latency before the next phase, or starting Churn only after Steady has settled, are methodological recommendations rather than runtime-enforced behavior of the framework. They are realised by the experimenter through scenario configuration and post-run analysis.

To illustrate how the methodology scales to third-party allocators, consider a hypothetical comparative study of jemalloc and mimalloc in the backend of a C++ chat server. The framework is natively driven by operation counts; the temporal durations and frequencies in the example below are illustrative and assume operation-count and tick-interval values chosen to produce them. The framework's actual experimental matrix evaluates native allocators (system malloc, monotonic_arena, pool_allocator, segregated_list), and OS-level effects such as madvise / MADV_DONTNEED behaviour are exposed only indirectly through the same memory footprint proxies used elsewhere in the methodology. A RampUp phase of 30 seconds ensures heap

filling by creating 50000 sessions of 10 KB each until a live set of 500 MB is reached, while its metrics are excluded from the final performance indicators. A Steady phase of 3 minutes at a frequency of 10000 operations per second keeps the live-set constant through the balance of session creation and destruction, making it possible to record the median latency of malloc and the p99 value, for example, 180 nanoseconds and 1.2 microseconds for jemalloc versus 150 nanoseconds and 600 nanoseconds for mimalloc.

A Churn phase of 2 minutes at a frequency of 80000 operations per second with random sizes from 64 bytes to 64 KB reveals the efficiency of free-block search under conditions of intensive rotation, where jemalloc p99 degrades to 4.5 microseconds due to traversal of size-class lists, whereas mimalloc keeps p99 at the level of 800 nanoseconds thanks to local thread caches. The final BulkReclaim phase, through instantaneous destruction of all sessions and subsequent RSS measurement, evaluates page return to the operating system, and if RSS remains at the level of 180 MB with a zero live-set, this testifies to the retention of freed pages in the internal pools of the allocator without invoking madvise with the MADV_DONTNEED flag.

Implementation note: Phase-level metric isolation applies fully to latency profiling and event counts. Volumetric trackers such as peakBytes are global by design, and their values reflect the cumulative state across the entire run rather than a per-phase window.

Measurement Model

The measurement system is based on an event-oriented architecture in which each act of interaction with the allocator generates a structured event. This allows separating the logic of workload execution from that of data collection, minimizing profiling overhead. The event stream E={e1, e2, …, en} includes phase start and end events, successful and unsuccessful allocation attempts, memory deallocation, and system ticks.

Metrics are defined as functions over the event stream: M = f (E). The methodology classifies measurements along several axes. Temporal metrics record phase-level and aggregated latency statistics, total phase duration, calculated throughput, median, and tail percentiles, rather than a per-operation latency log, since per-operation tracing would introduce unacceptable observer overhead. Counters track byte volumes and the number of calls. Memory-state metrics analyze live-set dynamics, peak consumption, and memory footprint proxies that approximate fragmentation (overhead ratio, reserved-to-live ratio). Particular attention is paid to time-series analysis. The tick mechanism enables periodic snapshots of the system state, sampled deterministically at a configurable operation-count interval (tickInterval) rather than at a wall-clock period, with timestamps attached to each tick to enable subsequent temporal analysis. These snapshots are necessary for identifying trends in performance degradation that are not visible in the final average values. Architecture of the measurement process shown in Figure 1.

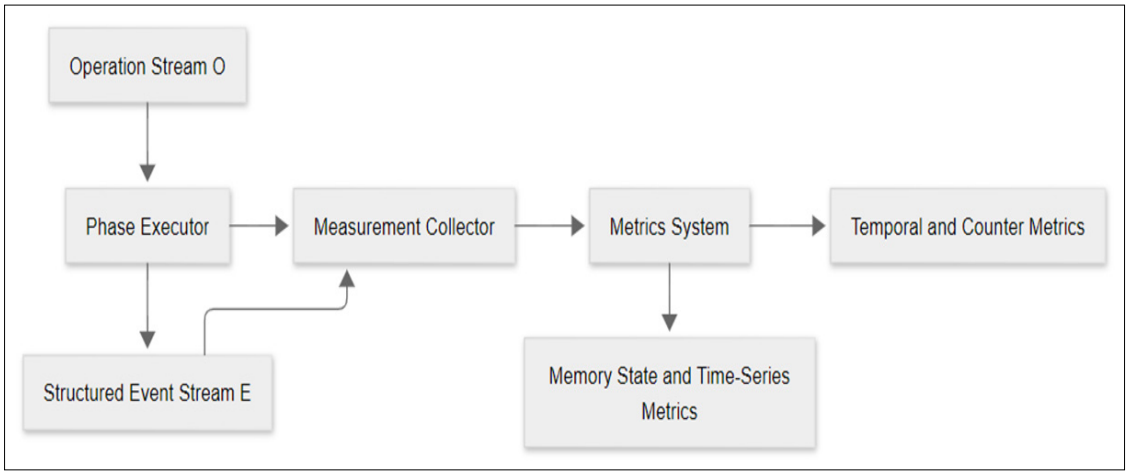

Fig. 1. Architecture of the measurement process

An important aspect is the handling of errors and undefined states. If an operation is not applicable to a given allocator type, for example, an attempt to resize in allocators with a fixed block size, the metric must be marked as N/A. The methodology prohibits equating N/A to zero, since this distorts the statistical significance of results when comparing different architectural solutions.

For the practical application of this model, the recommended baseline event set for each run should be defined first: phase start, phase completion, successful memory grant, failure, deallocation, and regular system pulse. The framework does not enforce this set as a hard runtime contract, emission of certain events, such as ticks or capability-probe events, depends on scenario configuration. To each emitted event, a scenario identifier, a timestamp, a memory-volume figure, and the aggregated volumetric state, live count, live bytes, peak bytes, maintained by the LifetimeTracker should be attached, so that, from a single stream, it becomes possible to reconstruct not only the average delay but also the moment at which degradation begins. Scenario identifiers are reliably

attached to summary outputs in CSV. Per-event JSONL serialization may omit redundant string identifiers to save bandwidth, with the binding to the scenario instead carried by the output stream as a whole.

In a real trial, this is conveniently done as follows: a service is launched with two types of memory allocators under an identical load of 100000 requests, the state is sampled every N operations via the configured tickInterval with timestamps attached to enable wall-clock-style analysis after the run, and operations that are invalid for a particular implementation are marked separately. Such cases should be preserved as not applicable and excluded from average calculations; otherwise, the comparison will show a false advantage. If, in terms of final average latency, both implementations are close, but periodic snapshots show growth of the live set and fragmentation only in one of them after the sixth minute of the test, the decision should be based on the time series, since it is the latter that reveals the future risk of failure under prolonged load.

CHAPTER 2. ENSURING REPRODUCIBILITY AND COMPARABILITY

In the field of systems research, repeatability of results is the principal criterion of scientific value. A large number of factors, from the OS scheduler to processor thermal throttling, can distort measurement results. This chapter presents mechanisms for controlling these factors.

Deterministic Execution Protocol

The methodology establishes the reproducibility invariant: for a fixed pair of parameters (P, S), the sequence of operations O must be constant, leading, in the absence of external noise, to identical metric values. To implement this protocol, the testing framework must provide full control over the seeds of pseudorandom number generators via a command-line interface.

An important condition is the isolation of allocations. Third-party allocations, for example, within the standard library or diagnostic tools, must not occur in the measured memory region. The methodology requires the use of separate heaps or arenas for the measurement framework's internal needs, so that its activity does not influence the state of the allocator under test. Otherwise, a heap contamination effect arises, in which the results of one test begin to depend on the order in which other tests are launched.

Implementation note: Full isolation is an architectural design goal. The LifetimeTracker supports an independent bufferAllocator for its tracking arrays, but when the system allocator malloc itself is the subject of the test, framework and test allocations necessarily share the same overarching memory space. The notion of a completely clean measured region is therefore an idealised condition, approached but not absolutely guaranteed across every scenario.

For this protocol to function in an applied environment, the same pair of input parameters and the same initial generator value should be fixed in advance for each trial, after which all internal service memory allocations should be moved into a separate region so that the measured segment remains completely clean.

In practical terms, this looks as follows: if two variants of a memory allocator are being tested for a protected service that processes 50 000 homogeneous authorization requests, a fixed initial value, for example 12345, should first be set through the command line; then logs, tracing, and internal buffers of the testing contour should be directed to a separate heap; after that, the same scenario should be run multiple times in the same order via the --repetitions CLI flag, the exact count is configurable; small repeat counts are typical for development, larger ones for the final report. If the sequence of operations coincides, but timing and memory consumption indicators begin to diverge between runs, the source of external interference or hidden service allocations within the measured region should be sought; this analysis is a manual interpretation step performed by the experimenter, since the framework does not currently include automated detection of such interference. It is precisely these factors that most often destroy reproducibility and render architectural comparison useless.

Control of Measurement Noise

Time measurement in modern multitasking environments requires accounting for system jitter. The methodology prescribes the use of a Timing Noise Control protocol that includes several stabilization levels. At the first level, warm-up iterations are performed, allowing the processor frequency to stabilize and the first- and second-level caches to fill with relevant data and instructions. At the second level, the experiment is repeated multiple times, and the median and 95th percentile are used as the final values, thereby excluding the influence of rare system interrupts.

To prevent compiler optimizations that can completely eliminate the measured code, the framework uses a custom internal DoNotOptimize barrier on values consumed by the measured operations. This barrier mitigates the risk of aggressive compiler elision; strict ordering at the level of out-of-order CPU execution is not guaranteed by the optimization barrier alone and would additionally require explicit hardware memory fences. It is also recommended to disable dynamic frequency scaling technologies, for example, Turbo Boost or SpeedStep, and to pin the thread to a specific physical processor core in order to minimize data migration among core caches. These hardware and OS configurations are external prerequisites for stable measurement; they are environmental recommendations rather than features automated by the methodology's software pipeline. The Timing Noise Control Protocol for Measurement Stabilization is shown in Figure 2.

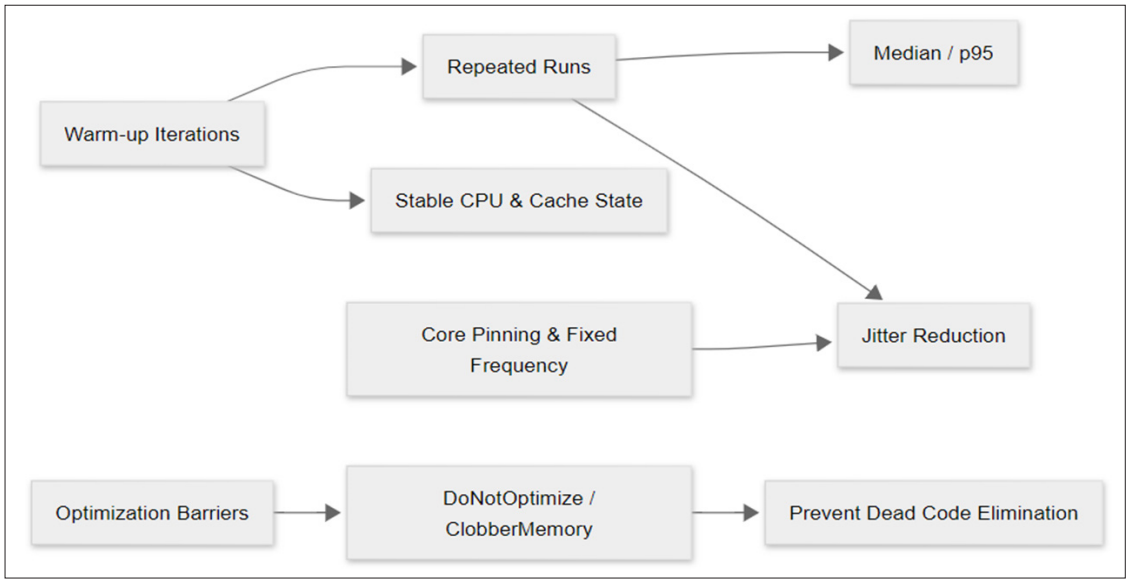

Fig. 2. Timing Noise Control Protocol for Measurement Stabilization

To illustrate, consider that the section of memory allocation and deallocation in an authentication verification contour is being evaluated. A run might begin with several hundred idle passes, for example, 500, then perform several dozen full series of operations, for example, 30 series of 10 000 operations, on one core, and only after that compare implementations. The specific values used here are illustrative. Warmup count, repetition count, operation count, and tick interval are configured per scenario via CLI flags, and default values for development workflows are typically considerably smaller. If the medians of two variants almost coincide, but the upper boundary for ninety-five percent of observations is sharply higher in one of them, the one with the narrower tail of the distribution should be chosen, since it is rare latency spikes that most often turn into failures under load.

Allocator Comparability Contract

For the correct juxtaposition of results from two different allocators, compliance with the comparability contract is required. Two allocators are considered comparable within the experiment if they execute the identical workload W1 = W2 with the same seed S and the same phase model E. Any violation of this contract, for example, the use of different object sizes or different workload intensity, renders the comparison meaningless. The reproducibility contract for Benchmark Validity is shown in Table 2.

Table 2. Reproducibility Contract for Benchmark Validity

Contract Element	Requirement	Rationale
Workload	Identity of P	Different distributions create different fragmentation conditions.
Generator (RNG)	Identity of S	The order of operations determines the physical placement of blocks.
Environment	Identity of HW/SW	Operating system and kernel versions affect the cost of system calls.
Metadata	Recording of build parameters	Compiler flags can change function latency.
NA handling	Separation of 0 and NA	It is invalid to treat non-applicability as zero cost.

The report on results must include the full environment profile: OS version, processor type, cache topology, compiler version, and build flags. Without these data, the results cannot be verified by external researchers. The methodology emphasizes that even minor changes in Linux kernel configuration, for example, the Transparent Huge Pages parameters, can produce tenfold differences in the speed of allocators that make intensive use of system calls.

In practical application, this may be done as follows: if a memory allocator is being chosen for a node that serves a stream of authentication requests, the same scenario of 200 000 operations with the same block sizes and identical order of actions should be set for both candidates, the trial should be carried out on one machine with the same build, and non-applicable operations should then be marked separately and not mixed with zero values. If, after this, one variant shows an advantage only under a different kernel configuration or with other build keys, such a result cannot be used to inform an architectural decision, since it reflects environmental factors masking the true properties of the memory allocation mechanism itself.

Implementation note: Operation counts, repetition counts, warmup iterations, and tick intervals are flexible CLI-configured inputs across the framework. Specific figures used in the example above, such as 200 000 operations, are illustrative. The actual experimental structure is parameterized end-to-end and can be adjusted per scenario without rebuilding the system.

CHAPTER 3. PRACTICAL GUIDE TO CONDUCTING EXPERIMENTS

The transition from theory to practice requires an understanding of the architectural principles underlying the construction of testing tools and the measurement methodology. This chapter describes the steps for deploying an experimental environment.

Architecture of the Benchmarking Framework

The architecture of a contemporary memory evaluation framework must be modular and extensible. The key components are ExperimentRunner, which is responsible for overall orchestration, and PhaseExecutor, which implements the logic for executing individual phases. The connecting link is EventBus, which ensures the delivery of event information from the executor to the measurement systems.

An important element is LifetimeTracker, a module that, based on the chosen lifetime model, determines which previously allocated objects must be deleted at a given time. This requires efficient management of the set of live pointers, which, in itself, is a task that requires optimization to ensure the tracker's overhead does not exceed the allocator's execution time. For these purposes, preallocated pools for storing metadata for live objects are recommended.

For the practical application of such an architecture, the testing contour should be immediately divided into independent modules, with one role assigned to each, so that new workload scenarios, object deletion rules, and metric-capture methods can be added without rebuilding the entire system. In real work, this is conveniently arranged as follows: one module launches the experiment and sets the order of phases, the second executes memory allocation and deallocation operations, the third transmits information about each action to the measurement system, and the lifetime-control module stores a preprepared array of references to active objects and, at the required moment, selects which object to delete according to the adopted model.

For example, if a stand is being prepared to test an authentication verification service under a load of 100 000 requests, a separate pool for service records of live objects with 120 000 positions should be allocated in advance so that accounting does not create superfluous delays, after which a new scenario with short and long phases can be connected without rewriting the rest of the system. Such an approach simplifies the evolution of the stand, keeps overhead under control, and allows comparisons without the hidden influence of the measurement contour.

Configuration and Launch of Experiments

Experiment configuration is performed through declarative interfaces. The methodology recommends using CLI flags to quickly adjust key parameters, such as --seed, --warmup, --repetitions, and --measurements. This enables integration of the testing process into continuous integration systems.

For each phase of the experiment, WorkloadParams parameters must be defined. For example, the RampUp phase may be configured to allocate small objects, 8–64 bytes, sequentially, whereas the Steady phase may use a Pareto distribution to simulate realistic data rotation. Registration of experiments in the framework must be automated so that the researcher can launch entire families of tests with a single command.

To turn such a configuration into a working tool, a set of typical runs should be assembled in advance, and all key parameters should be specified through the command line, so that the same experiment can be repeated without manual editing locally, in the build contour, and on a dedicated test node. Practically, this is done as follows: for the initial growth phase, sequential allocation of blocks from 8 to 64 bytes is specified; for the steady phase, a mixed size distribution with a predominance of small objects and rare large spikes is set; then a fixed initial generator value, the number of warm-up passes, the number of repetitions, and the measurement volume are preserved in one launch profile. Recommended experiment configuration and execution profile are shown in Table 3.

Table 3. Recommended Experiment Configuration and Execution Profile

Configuration Element	Recommended Setting	Practical Purpose
CLI parameters	--seed, --warmup, --repetitions, --measurements	Ensures fast, reproducible, and automation-friendly experiment setup
RampUp phase	Sequential allocation of 8–64 byte objects	Simulates controlled initial memory growth
Steady phase	Mixed size distribution with mostly small objects and rare large spikes	Approximates realistic data rotation patterns
Registration	Automated experiment registration	Enables one-command execution of whole test families
Execution profile	Fixed seed, warmup count, repetition count, and measurement volume	Supports reproducible runs across local, CI, and dedicated environments

For example, if memory is being tested for an authentication service, a family of three scenarios should be created with a single command: a short load for rapid checking, a medium one for daily control, and a long one for degradation search.

Then, upon each change in code or environment parameters, it will immediately become visible in which phase latency growth begins, and the workload profile can be modified consciously while preserving the comparability of all subsequent results.

Typical Experimental Scenarios

The methodology proposes a set of standard experimental scenarios that must be checked for any new allocator. The Fixed-size scenario assumes the allocation and deallocation of blocks of one size. Such a regime is used to evaluate the efficiency of memory pool operations.

The Variable-size scenario is oriented toward operation with objects of different sizes. It enables assessing how efficiently block-splitting and merging mechanisms are implemented. This is especially important for analyzing allocator behavior under heterogeneous load conditions. The Bounded Churn scenario describes long-term operation under limited memory volume. Its application enables investigation of the effect of gradual fragmentation accumulation. Such a regime helps to understand how the state of memory changes during prolonged exploitation.

The Bulk Reclaim scenario is intended to verify the speed of memory cleanup at the completion of major logical stages in software operation. Such a scenario shows how quickly the system is able to release significant volumes of previously occupied memory.

The phase composition RampUp → Steady → Churn → Reclaim is regarded as the gold standard of comprehensive analysis. It enables tracing the memory life cycle from initial acquisition by the operating system to final resource release. Such an approach helps reveal anomalies arising at each stage of memory work.

To use these scenarios profitably in development, each should first be correlated with a real class of tasks in the system, and a new allocation method should be tested only against the set of phases that reflect future operation. For message queues and homogeneous structures, a fixed-block-size regime should be used. For mixed business logic involving both small and large objects, a variable-size regime should be adopted. For long-running services under a strict memory limit, prolonged shuffling with a bounded volume must be conducted. For batch document processing or completion of large computational stages, bulk reclamation should be measured separately.

A concrete example is as follows: if an order-processing server is being designed, memory should first be filled with a series of initial allocations, then the system should be moved into a steady phase with alternation of 32-, 128-, and 2048-byte objects, after that a constant memory ceiling should be maintained for 20 minutes, and at the end the entire large data set should be freed in one wave. If, after this, allocation time grows from minute to minute and memory return at the end takes too long, then the chosen strategy will create a risk of delays already in the production contour and must be replaced before deployment.

Analysis of Results and Interpretation

Output data must be presented in two formats: JSONL for detailed time-series analysis and CSV for summary tables. Analysis of results includes examining the shape of the latency distribution. The presence of high p99 values often indicates periodic blocking or expensive operations of reorganizing the allocator's internal structures.

Efficiency criteria include several key indicators that enable quantifying the behavior of the memory management system. Latency reflects the average and peak execution time of operations. This indicator allows judging allocator speed under normal conditions and during peak load.

Fragmentation, in the methodology's measurement model, is approximated through the overhead ratio of reserved to live bytes; direct heap-introspection-based fragmentation is not exposed by the framework. This proxy shows how rationally the available address space is used and the extent to which internal memory-distribution processes inflate reserved memory beyond what is actually live.

Peak consumption characterizes the maximum amount of memory acquired from the operating system. This parameter is important for understanding the upper bound of resource costs arising during application execution.

Cache efficiency, where instrumented, is expressed through the number of cache misses per operation. The current framework does not directly capture hardware performance counters; integration of cache-miss instrumentation is listed in §4.2 as planned work. Where it is available, this indicator allows assessing how well memory organization aligns with the characteristics of the processor's memory hierarchy.

The interpretation of data must take task specificity into account. In real-time systems, a low median is less important than the absence of outliers in p99. At the same time, for batch data processors, overall throughput is important, even if individual operations take a long time.

CHAPTER 4. POSITIONING AND LIMITATIONS OF THE METHODOLOGY

For the correct application of the methodology, it is necessary to understand its place among existing tools and the limitations imposed by the adopted model.

Comparison with Existing Approaches

Existing evaluation methods often focus on narrow aspects, which does not allow for a holistic picture. A comparison of the proposed methodology with classical approaches is presented in the table below. Comparative analysis of memory allocator benchmarking methodologies is shown in Table 4.

Table 4. Comparative Analysis of Memory Allocator Benchmarking Methodologies

Parameter	Wilson, Trace-driven	Larson, Throughput	Proposed Methodology
Workload Type	Static traces	Synthetic stress	Generative phase models
Lifetime Handling	Implicit	Absent	Explicit, parameterized
Reproducibility	High	Medium	Extreme, invariant S
Fragmentation Analysis	Primary focus	Absent	Integrated through time series
Scalability	Limited by trace	High	Adjustable through P

Unlike Wilson's approach, which depends heavily on the quality and timeliness of collected traces, the proposed methodology enables investigation of what-if scenarios by changing distribution parameters or lifetime policies without the need to recollect data in production. Compared with the Larson test, the methodology provides a much deeper understanding of the causes of performance decline by linking them to specific phases of memory exploitation.

Limitations and Directions for Development

The methodology has known limitations, several of which delineate scope rather than indicate gaps in the current implementation. They are listed below alongside planned framework extensions and items addressed in companion publications of the same research line.

The methodology isolates allocator behavior from concurrency effects; multithreaded extensions are out of scope for the present work and form a separate axis of the research line. Although many principles, determinism, phase structure, the comparability contract, generalise naturally, the effects of false sharing and lock contention require additional measurements not yet integrated into the model. The workload W is an approximation of real program behavior; distribution parameters require periodic validation against production profiles, and replay of captured traces is not currently supported.

Memory footprint proxies in place of direct OS-level metrics. RSS, the fragmentation coefficient, and madvise-related effects are approximated through reserved-byte, live-byte, and peak-byte counters maintained by the LifetimeTracker, and through the derived overhead ratio. Direct OS-level introspection, for example, via /proc/<pid>/status or getrusage, is not exposed by the framework. Mixed-lifetime effects are studied by comparing isolated runs of single-policy configurations rather than by executing multiple lifetime models simultaneously within a single workload.

Anomalies caused by external interference, missing isolation, or hidden allocations are surfaced through cross-run comparison and post-run analysis rather than by automated detection in the framework itself. CPU pinning, dynamic-frequency-scaling control, and similar measurement-stability configurations are environmental recommendations rather than features automated by the framework.

Direct OS-level memory metrics include integration of RSS sampling based on /proc/<pid>/status or getrusage, together with probes of madvise behaviour. These metrics complement the existing footprint proxies and provide a more direct view of memory use at the operating-system level. Simultaneous mixed-lifetime workloads refer to a unified execution mode where several lifetime models run at the same time within one workload. This makes it possible to measure cross-policy fragmentation effects directly, because different allocation lifetimes interact inside the same execution environment.

Built-in stability gates and termination predicates provide first-class support for phase transitions based on p99 latency, throughput, and reserved bytes. They also support open-ended termination conditions, which helps describe workload phases and stopping criteria more systematically. Hardware performance counter integration adds cache-miss, TLB-miss, and branch-misprediction sampling to the measurement model. These counters expand the analysis beyond memory footprint and allocation behaviour by adding hardware-level performance signals.

Automated reproducibility diagnostics use cross-run drift detection to reveal external interference and hidden service allocations. This reduces the need for manual investigation when results vary across repeated runs. Machine learning can be used for allocator parameter selection on a given workload. This approach builds on the parameterized workload model and supports more systematic exploration of allocator configurations.

Beyond the extensions already enumerated, four further directions delineate the planned trajectory of the framework and address dimensions of allocator behavior left outside the scope of the present formulation. NUMA-aware allocation and memory locality modeling constitute the first of these. The current methodology isolates allocator behavior on a single processor socket, which sets aside the cost asymmetry of remote memory access, the placement of arenas relative to executing threads, and the migration of pages between nodes under load. The planned extension introduces a node-aware variant of the workload model in which operations carry an originating-node attribute, the LifetimeTracker maintains per-node live-set counters, and the comparability contract is widened to fix the topology of cores and memory controllers across runs. Interaction with the virtual memory subsystem forms the second direction. Paging behavior, kernel overcommit policies, and the use of transparent or explicit huge pages exert pronounced influence on the cost

of allocation operations and on the relationship between reserved bytes and resident set size, yet they remain outside the current measurement model that operates at the level of allocator-internal counters. The planned instrumentation samples major and minor page fault counts, tracks transitions between page sizes, and records the disposition of pages returned through madvise, which permits attribution of latency tails to specific virtual memory events rather than to the allocator alone.

Validation of workload models against production traces constitutes the third direction and addresses a structural limitation of generative experimentation. The parametric space P approximates real program behavior through statistical distributions, which leaves open the question of fidelity to deployed systems. The planned validation procedure ingests captured allocation traces from production environments, fits the parameters of P to the observed distributions of sizes, alignments, and lifetimes, and applies divergence measures over the resulting distributions to quantify the residual gap between synthetic and recorded workloads. Trace replay is added as a complementary execution mode that reconstructs the operation stream O from a captured log under the same phase and measurement infrastructure, which permits direct comparison between generative and trace-driven runs on identical allocators. Alignment and cache-line effects form the fourth direction and complement the planned hardware performance counter integration.

The current alignment distribution within WorkloadParams determines the alignment requirement of individual blocks without modeling the spatial relationship between allocated objects and the cache geometry of the target processor. The planned extension records the cache-line footprint of each allocation, samples the co-residency of objects with differing lifetime policies within the same line, and reports indicators of locality degradation and false sharing under workloads that interleave short-lived and long-lived objects within shared cache lines. Together, these four extensions preserve the central operating principles of determinism, phase decomposition, and event-driven measurement while widening the methodology toward dimensions of memory behavior that govern the performance of contemporary C++ systems on multi-socket hardware with deep memory hierarchies.

CONCLUSION

The developed methodology formalizes the experimental evaluation of memory allocation strategies in C++ systems by combining a generative workload, an explicit specification of object lifetime, phase-based execution decomposition, and an event-driven measurement model. Such a construction transfers allocator analysis from the plane of disparate microtests into the regime of reproducible research, where performance, fragmentation, peak memory consumption, and live-set dynamics are treated as interrelated characteristics of a single system. The introduction of object lifetime as the primary parameter of the experiment reveals the nature of degradation arising from the mixing of different memory placement and reclamation patterns.

The proposed reproducibility protocol establishes strict conditions for allocator comparison by requiring identical workloads, generator seeds, phase schemes, hardware, and software environments, as well as separate accounting for non-applicable operations. By controlling system noise, adjusting build parameters, and separating service allocations for the measurement contour, the methodology lays a foundation for verifiable results suitable for both academic analysis and engineering decision-making. The experimental design allows for the comparison of warm-up cost, behavior in the steady-state regime, resilience to churn load, and the quality of bulk memory returned to the operating system.

The practical value of the work lies in the possibility of designing memory architecture based on observable workload properties and lifetime models, including scenarios with fixed and variable object sizes, prolonged memory limitations, and bulk resource release. The methodology occupies a distinctive place among existing approaches due to its parameterized description of lifetime and its built-in time-series fragmentation analysis. Its current limitation is its emphasis on single-threaded execution, which leaves open the development of the model toward multithreading, NUMA systems, and hardware counters while preserving the central operating principle.

REFERENCES

1. Bailleu, M., Stavrakakis, D., Rocha, R., Chakraborty, S., Garg, D., & Bhatotia, P. (2024). Toast: A Heterogeneous Memory Management System. *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*, 53–65. https://doi.org/10.1145/3656019.3676944
2. Bell, O., Kumar, A., & Gill, C. (2024). Host-Based Allocators for Device Memory. *ArXiv*. https://doi.org/10.48550/arXiv.2405.07079
3. Cuadros Zegarra, E., Barrios Aranibar, D., & Cardinale, Y. (2024). IoRT-Based Middleware for Heterogeneous Multi-Robot Systems. *Journal of Sensor and Actuator Networks*, *13*(6), 87. https://doi.org/10.3390/jsan13060087
4. Hong, J., Jang, W., Kim, M., Yu, L., Kwon, Y., & Jeon, Y. (2025). CMASan: Custom Memory Allocator-Aware Address Sanitizer. *Proceedings of 2025 IEEE Symposium on Security and Privacy*, 740–757. https://doi.org/10.1109/sp61157.2025.00110
5. Li, R., Wu, Q., Kavi, K., Mehta, G., Beard, J. C., Yadwadkar, N. J., & John, L. K. (2025). SpeedMalloc: Improving Multi-threaded Applications via a Lightweight Core for Memory Allocation. *ArXiv*. https://doi.org/10.48550/arXiv.2508.20253

6. Maas, M., Andersen, D. G., Isard, M., Javanmard, M. M., McKinley, K. S., & Raffel, C. (2024). Combining Machine Learning and Lifetime-Based Resource Management for Memory Allocation and Beyond. *Communications of the ACM*, *67*(4), 87–96. https://doi.org/10.1145/3611018

7. Nam, M. J. (2024). FRAMER/Miu: Tagged Pointer-based Capability and Fundamental Cost of Memory Safety & Coherence. *ArXiv*. https://doi.org/10.48550/arXiv.2408.15219

8. Zhou, Z., Gogte, V., Vaish, N., Kennelly, C., Xia, P., Kanev, S., Moseley, T., Delimitrou, C., & Ranganathan, P. (2024). Characterizing a Memory Allocator at Warehouse Scale. *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 192–206. https://doi.org/10.1145/3620666.3651350