

Eötvös Loránd University
Faculty of Informatics
Department of Programming Languages and Compilers

Migrating Spring Boot MVC Systems to Java 21 Virtual Threads: Architecture, Risks, and Engineering Guidelines

Author:

Hamed Mohamed

Computer Science MSc — Software Architecture Specialisation

Neptun code: DG1EWF

Internal supervisor:

Dr. habil. Tamás Kozsik

Associate Professor, Head of Department

External supervisor:

Gregory Reynolds Morse

Lecturer and MSc in Computer Science

Budapest, 2026

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

Thesis Topic Registration Form

Student's Data:

Student's Name: Hamed Mohamed

Student's Neptun code: DG1EWF

Educational Information:

Training programme: programtervező informatikus, mesterképzés (MA/MSc)

I have an internal supervisor

Internal Supervisor's Name: Gregory Reynolds Morse

Supervisor's Home Institution: **Department of Programming Languages and Compilers**

Address of Supervisor's Home Institution: **1117, Budapest, Pázmány Péter sétány 1/C.**

Supervisor's Position and Degree: Lecturer and MSc in Computer Science

Track within Programme:Computer Science (Software Architecture)

Thesis Title: Migrating Spring Boot MVC Systems to Java 21 Virtual Threads: Architecture, Risks, and Engineering Guidelines

Topic of the Thesis:

(Upon consulting with your supervisor, give a 150-300-word-long synopsis os your planned thesis.)

Modern enterprise Java applications rely on synchronous request-response patterns with thread pools to handle concurrent workloads. Java 21 Virtual Threads (Project Loom) promise orders of magnitude higher throughput by enabling millions of lightweight threads, but adoption in legacy Spring Boot MVC systems is fraught with risks. Existing codebases contain architectural anti-patterns—synchronized blocks, ThreadLocal misuse, unbounded blocking I/O—that cause thread pinning, completely negating Virtual Thread benefits. This thesis proposes a systematic approach to diagnose and migrate such systems safely.

The project constructs a realistic Spring Boot MVC demonstration application intentionally embedding common anti-patterns found in enterprise code. Using comprehensive benchmarking (k6 load testing, Java Flight Recorder pinning detection), we quantify performance impact across three modes: Platform Threads baseline, Naive Virtual Thread adoption (–63% throughput), and Refactored Virtual Threads (+96% recovery). Analysis focuses on identifying pinning sources via JFR stack traces and correlating them with architectural issues.

We evaluate this framework across realistic workloads (50, 100, 200 concurrent users) measuring throughput, latency percentiles, and error rates. Results demonstrate that naive adoption worsens performance; systematic refactoring (synchronized → ReentrantLock, ThreadLocal auditing, connection pool tuning) recovers baseline performance and enables scalability.

The expected outcome is a practical 6-step migration checklist, a catalog of anti-patterns with quantified impacts, and a reproducible benchmarking suite for diagnosing pinning in legacy Spring Boot systems.

Budapest, 2026. 07. 02.

Abstract

This dissertation presents an empirical systems investigation into the architectural, performance, and operational implications of migrating enterprise Spring Boot Model-View-Controller (MVC) applications to Java 21 Virtual Threads (Project Loom, JEP 444). While virtual threads decouple Java threads from operating system kernel threads to eliminate web-tier thread pool exhaustion, this research investigates the *Schedulability Fallacy*: the widespread industrial misconception that user-mode thread multiplexing automatically resolves concurrency bottlenecks without regard to domain synchronization, queueing equilibrium, downstream resource constraints, or foreign-function boundaries.

To evaluate these phenomena, a controlled Spring Boot 3.5.12 MVC demonstration application was developed and subjected to 125 physical benchmark executions across eight non-overlapping experimental strata (including the Two-Treatment Factorial Persistence design in Stratum 8). The experimental corpus—encompassing raw k6 JSON lines streams, Java Flight Recorder (JFR) binary event traces, and operating system telemetry—was forensically audited and cataloged; the recorded normalized comparisons were SHA-256 verified. Workloads were analyzed using a dual-window reporting standard that separates transient ramp-up artifacts from steady-state queueing equilibrium.

The empirical findings establish that:

1. In the tested Java 21 condition, enclosing blocking operations within intrinsic monitors (`synchronized`) produced qualifying `jdk.VirtualThreadPinned` events. Replacing `synchronized` with a virtual-thread-compatible `ReentrantLock` produced zero qualifying events in the cited comparison while yielding similar throughput (1.956 vs. 1.967 req/s), consistent with the measured single-server serialization ceiling ($\mu \approx 2.0$ req/s).

2. Non-fair `ReentrantLock` showed a descriptive approximately 23-fold difference in sample standard deviation ($\sigma = 16.53$ ms vs. 0.70 ms) compared to `synchronized`, revealing a possible tail-latency trade-off under the tested conditions.

3. In the tested direct Java Native Interface (JNI) treatment, blocking POSIX `nanosleep()` occupied a carrier in the observed Darwin/Mach wait state with low process CPU ($1.04\% \pm 0.12\%$) and **zero qualifying JFR pinning events**, exposing a scoped observability gap.

4. In relational persistence, holding connections across blocking operations empirically validates the Downstream Bottleneck Shift Law: virtual threads unmount cleanly without pinning carrier threads (0 JFR events), but throughput is strictly capped by pool size ($X_{\max} = K/T_{\text{hold}}$), shifting queue accumulation entirely into the connection pool.

Based on these discoveries, this dissertation formalizes the *Three-Tier Concurrency Taxonomy* and delivers an authoritative engineering migration framework comprising five mandatory architectural rules, four correctness-preserving refactoring patterns, and a deterministic migration decision tree.

Keywords: Java 21, Virtual Threads, Project Loom, Concurrency Engineering, Software Architecture.

Kivonat

Jelen diplomamunka a Java 21 virtuális szálak (Project Loom, JEP 444) vállalati Spring Boot Model-View-Controller (MVC) alkalmazásokban történő bevezetésének architekturális, teljesítménybeli és üzemeltetési hatásait vizsgálja empirikus rendszerszintű megközelítéssel. Bár a virtuális szálak szétválasztják a Java szálakat az operációs rendszer kernelfelületű platformszálaitól – ezzel megszüntetve a webes konténerek szálkészlet-kimerülési korlátait –, a kutatás részletesen cáfolja az iparági *Ütemezhetőségi Tévedést* (Schedulability Fallacy). Ez a tévedés azon a hibás feltételezésen alapul, hogy a felhasználói szintű szálmultiplexelés automatikusan feloldja a párhuzamossági szűk keresztmetszeteket anélkül, hogy figyelembe venné a szinkronizációt, a sorbanállási egyensúlyt, a perzisztens adatbázis-erőforrások korlátait vagy a natív futási határokat.

A jelenségek vizsgálatához egy Spring Boot 3.5.12 MVC demonstrációs alkalmazást fejlesztettünk, amelyet 125 fizikai mérésnek vetettünk alá nyolc, egymástól elkülönített kísérleti rétegben (beleértve a 8. réteg kétkezeléses faktoriális perzisztencia-vizsgálatát). A keletkezett adatbázis – amely nyers k6 terhelési naplókat, Java Flight Recorder (JFR) bináris eseményfájlokat és operációs rendszer telemetriát tartalmaz – teljes körű törvényszéki vizsgálaton esett át, és egy nyílt tudományos archívumban került rögzítésre; a normalizált összehasonlítások rögzített SHA-256 ellenőrzőösszegei egyeztek.

Az empirikus eredmények az alábbiakat igazolják:

1. Java 21 futtatókörnyezetben a blokkoló műveletek belső monitorba (`synchronized`) zárása determinisztikusan a hordozószálhoz rögzíti a virtuális szálat (thread pinning), és `jdk.VirtualThreadPinned` eseményeket generál. Ugyanakkor a `synchronized` kiváltása egy virtuálisszál-biztos `ReentrantLock` primitívvel teljesen megszünteti a rögzítési eseményeket (108-ról 0-ra), ám **nem eredményez átviteli sebességnövekedést** (1.956 vs. 1.967 req/s), mivel a kritikus szakasz Amdahl-törvénye által diktált kiszolgálási korlátja ($\mu = 1/D = 2.0$ req/s) változatlan marad.

2. A nem-igazságos `ReentrantLock` esetén a minta szórása leíró jelleggel hozzávetőleg 23-szor nagyobb volt ($\sigma = 16.53$ ms vs. 0.70 ms), mint a `synchronized` blokk esetén; ez a vizsgált feltételek között lehetséges faroklatencia-kockázatra utal.

3. A közvetlen Java Native Interface (JNI) hívásokon keresztül végrehajtott blokkoló POSIX rendszerhívások némán fogva tartják a hordozószálat Darwin/Mach várakozási állapotban: a CPU-kihasználtság minimális marad ($1.04\% \pm 0.12\%$), miközben **egyetlen JFR rögzítési esemény sem keletkezik**, elfedve a szálkiéheztetést a hagyományos megfigyelőrendszerek elől.

4. A relációs perzisztencia terén a kapcsolatok blokkoló műveletek alatti megtartása empirikusan igazolja a lefelé tolódó szűk keresztmetszet törvényét (Downstream Bottleneck Shift Law): a virtuális szálak tisztán lekapcsolódnak a hordozószálról rögzítés nélkül (0 JFR esemény), ám az átviteli sebességet szigorúan korlátozza a kapcsolatkészlet mérete ($X_{\max} = K/T_{\text{hold}}$), így a sorbanállás teljes egészében a kapcsolatkészletbe tevődik át.

Ezen eredményekre építve a dolgozat felállítja a *Háromszintű Párhuzamossági Taxonómiát*, és egy mérvadó szoftverarchitektúrás migrációs keretrendszert biztosít, amely öt kötelező érvényű szabályt, négy helyességmegőrző refaktorálási mintát és egy determinisztikus döntési fát tartalmaz a gyakorló mérnökök számára.

Kulcsszavak: Java 21, Virtuális Szálak, Project Loom, Concurrency Engineering, Szoftverarchitektúra.

Contents

1 Introduction **5**

1.1 Programme and Specialisation Fit 5

1.2 Motivation and Research Problem 6

1.2.1 The Virtual Thread Bottleneck Fallacy 7

1.3 Research Goals, Questions, and Hypotheses 8

1.4 Scope, Assumptions, Constraints, and Exclusions 10

1.4.1 Explicit Methodological Exclusions 11

1.5 Contributions and Responsibility Boundaries 12

1.5.1 The Three Foundational Architectural Principles 14

1.6 Thesis Roadmap 14

2 Related Work **16**

2.1 Review Scope and Source Strategy 16

2.2 Five-Domain Concurrency Taxonomy 17

2.3 Closest Prior Work and Comparative Delta Matrix 23

2.4 Critical Synthesis and the Research Gap 24

2.4.1 The Identified Research Gap 25

2.5 Positioning of the Proposed Contribution 26

3 Background and Foundations **27**

3.1 Evolution of Concurrency in the Java Virtual Machine 27

3.2 Project Loom and Virtual Thread Architecture 28

3.2.1 Continuation Abstraction and Chunked Stack Management 28

3.2.2 The Work-Stealing Carrier Scheduler 29

3.3 Mechanics of Monitor Pinning in Java 21 29

3.3.1 HotSpot C++ Continuation Freeze Mechanics 29

3.3.2 The JDK Runtime Evolution: JEP 491 in Java 24 30

3.4 Synchronization Primitives: Intrinsic Monitors vs. Explicit Locks 31

3.4.1 Language-Level Intrinsic Monitors (`synchronized`) 31
3.4.2 Explicit Locks and the AbstractQueuedSynchronizer (AQS) 31
3.4.3 The NonfairSync Barging Anomaly 32
3.5 Persistence Concurrency: JDBC and Connection Pools 32
3.5.1 HikariCP Connection Pool Mechanics 32
3.5.2 The Downstream Bottleneck Shift Law 33
3.6 ThreadLocal Semantics and Request Context 33
3.7 Foreign Function Boundary: JNI and Native Execution 34
3.7.1 Carrier Captivity in Native Downcalls 34
3.7.2 The Observability Vacuum 34
3.8 Observability, Diagnostics, and Telemetry 35
3.9 Performance Foundations and Causal Queueing Models 35
3.9.1 Little's Law and Closed-Loop Systems 35
3.9.2 Amdahl's Law and Single-Server Serialization 36

4 Research Design and Methodology 37
4.1 Research Design and Mode 37
4.2 Question-to-Method Operationalisation 38
4.3 Materials, Data, Tools, and Provenance 40
4.3.1 Repository Evidence Strata Census 40
4.4 Workload Protocol and Closed-Loop Queueing Mechanics 41
4.4.1 Closed-Loop vs. Open-Loop Mechanics 42
4.5 Dual-Window Steady-State Analysis Standard 42
4.6 Telemetry Pipeline and Correctness Invariants 43
4.7 Validity Controls and Methodological Mitigations 44
4.8 Academic Ethics and AI Disclosure 44

5 Implementation and Research Infrastructure 45
5.1 System Architecture and Application Anatomy 45
5.1.1 Framework Bill of Materials and Pinned Dependencies 46
5.2 Tomcat Request Dispatch and Virtual Thread Wiring 47
5.3 Payment Synchronization Implementations 47
5.4 Foreign Function Interface: Native POSIX Shared Library 48
5.4.1 HotSpot Carrier Captivity and Diagnostic Omission 48
5.5 Persistence Layer and HikariCP Connection Pooling 49

5.5.1 Two-Treatment Factorial Persistence Design 49
5.6 Request Context and ThreadLocal Lifecycle 50
5.7 Benchmarking Harness and Telemetry Pipeline 50
5.8 Verification Suite and Artifact Provenance Tooling 51

6 Experimental Results **52**
6.1 Evidence Inventory and Provenance Ledger 52
6.1.1 Semantic Invariant and Correctness Verification 53
6.2 Results for RQ1: Synchronization, Pinning, and Serialization 54
6.2.1 Key RQ1 Empirical Invariants 54
6.2.2 Trial-Level Dispersion and the AQS Barging Signature (CLM-RQ1-05) 55
6.2.3 Java 24 Runtime Evolution: JEP 491 Unpinning (Stratum 2) 55
6.3 Results for RQ2: Architectural Migration Risks 56
6.3.1 Foreign Function Interface: JNI Native Boundary Results (Stratum 3) 56
6.3.2 Two-Treatment Factorial Persistence Ablation (Stratum 8) 56
6.3.3 Supporting Architectural Controls (Stratum 4) 57
6.4 Historical Baselines and Flawed Strata (Strata 5 and 7) 57
6.5 Negative, Null, and Ablation Findings 58

7 Analysis, Discussion, and Migration Guidance **59**
7.1 Explicit Answers to the Research Questions 59
7.2 The Three-Tier Concurrency Taxonomy 61
7.3 The Authoritative Migration Framework 62
7.3.1 The Five Core Architectural Migration Rules 62
7.3.2 Correctness-Preserving Refactoring Patterns 63
7.3.3 Master Migration Decision Matrix and Decision Tree 64
7.4 Master Threats to Validity 65
7.4.1 Transferability Boundary Audit 66
7.5 Engineering, Operational, and Academic Relevance 67
7.5.1 Architectural and Operational Relevance 67
7.5.2 Target Peer-Reviewed Venues 67
7.5.3 Cumulative Research Protocol: Java 24 GA Replication Roadmap . 67

8 Conclusion **69**

8.1 Answers and Achieved Goals 69

8.2 Contribution Summary 70

8.3 Core Engineering Lessons and Paradigm Shift 71

8.4 Limitations and Boundary Conditions 71

8.5 Future Research Directions 72

Bibliography **74**

List of Tables **80**

A Reproduction and Independent-Check Record **82**

A.1 Reproduction Overview and Open-Science Package 82

B Ethics, Data, Licensing, and AI Record **85**

B.1 Research Governance and Compliance Overview 85

B.2 Detailed Disclosure of Generative AI Assistance 86

B.2.1 Strict Exclusions and Independent Authorship Guarantees 87

C Process and Submission Checklist **89**

C.1 Master Regulatory and Submission Verification 89

Chapter 1

Introduction

1.1 Programme and Specialisation Fit

This master's dissertation is submitted in partial fulfillment of the requirements for the degree of *Master of Science in Computer Science* within the *Software Architecture Specialisation* at the Faculty of Informatics, Eötvös Loránd University (ELTE), Budapest [13]. The research was conducted under the institutional auspices of the Department of Programming Languages and Compilers, under the joint supervision of Associate Professor Dr. habil. Tamás Kozsik (Head of Department) and external thesis advisor Gregory Reynolds Morse (Lecturer and MSc in Computer Science) [33], pursuant to approved MSc thesis topic registration `IK-ZVMSC_EN2_155`.

The dissertation directly addresses three core subject areas of the MSc curriculum and final examination syllabus:

1. **Software Architecture and Scalable Enterprise Systems:** Evaluating request-dispatching topologies, concurrency bounding, connection-pool queueing dynamics, and architectural boundary preservation when migrating synchronous enterprise web services from OS-managed thread pools to lightweight user-space threads.

2. **Advanced Java Programming and Runtime Internals:** Deconstructing Java Virtual Machine (JVM) execution mechanics, continuation capture, intrinsic monitor synchronization (`synchronized`), explicit lock synchronization (`java.util.concurrent.locks.ReentrantLock`), the AbstractQueuedSynchronizer (AQS) framework, and the OpenJDK HotSpot runtime transition from Java 21 LTS (JEP 444 [40]) to Java 24 (JEP 491 [41]).

3. **Systems Engineering, Project Management, and Empirical Rigour:** Designing reproducible benchmark suites, managing experimental artifacts, instituting open-science data freezes with cryptographic integrity, and establishing compliance with academic governance standards, including ELTE Dean's Directive 1/2026 [12].

1.2 Motivation and Research Problem

For more than two decades, enterprise Java backend architectures have overwhelmingly relied upon the synchronous, *thread-per-request* paradigm. Standardized by the Java Servlet API and realized in production web servers such as Apache Tomcat [3] and Spring Framework Model-View-Controller (MVC) containers [45], this architecture assigns each incoming HTTP connection to a dedicated worker thread. This paradigm provides immense developer ergonomics: sequential imperative code execution, natural diagnostic stack traces, implicit request-scoped context propagation via `ThreadLocal`, and seamless integration with synchronous blocking persistence APIs such as JDBC and HikariCP [49].

However, this architecture historically rested upon an inflexible foundation: a strict $1:1$ mapping between Java threads (`java.lang.Thread`) and operating system (OS) kernel platform threads (such as POSIX `pthread`s). Because platform threads allocate heavyweight virtual memory stacks (defaulting to 1 024 KB per thread in HotSpot via `-Xss1m`) and incur preemptive kernel context-switching latency, enterprise containers strictly bound concurrency using fixed thread pools (e.g., Tomcat's default `maxThreads = 200`). Under high concurrency or during downstream I/O latency spikes, worker thread pools exhaust rapidly, inducing connection backlog queueing, exponential latency escalation, and catastrophic server timeouts.

To circumvent platform thread exhaustion, reactive, event-driven architectures (such as Spring WebFlux, Project Reactor, and Netty) emerged during the 2010s. By multiplexing concurrent connections across a tiny pool of non-blocking event-loop threads matching CPU core counts, reactive systems eliminated thread pool exhaustion. Yet this efficiency imposed severe cognitive and operational costs: the abandonment of imperative control flow in favor of monadic functional pipelines (`Mono`, `Flux`), fragmented stack traces that obscure production debugging, and fundamental architectural friction when interfacing with blocking relational database drivers (JDBC).

The delivery of **Project Loom** in **Java 21 LTS** via **JEP 444** [40] was engineered to eliminate this trade-off. Virtual Threads introduce lightweight, user-mode threads managed directly by the JVM runtime under an $M : N$ scheduling architecture:

- Millions of ephemeral virtual threads (M) can be instantiated with minimal initial memory footprints ($\approx$ 1 KB), storing their call frames as `Continuation` objects directly on the garbage-collected Java heap.
- A small pool of OS platform threads (N), managed by a work-stealing HotSpot `ForkJoinPool` (`DEFAULT_SCHEDULER`), acts as *carrier threads* that physically execute instructions on CPU cores.
- When a virtual thread executes a blocking operation within standard Java libraries (such as socket I/O, `Thread.sleep()`, or `ReentrantLock.lock()`), the runtime captures its continuation frames to the heap and *unmounts* the virtual thread from its carrier, freeing the underlying OS thread to execute other ready virtual threads.

In Spring Boot 3.2.0 and later (evaluated in this dissertation on Spring Boot 3.5.12), enabling virtual threads requires only a single configuration property:

```
spring.threads.virtual.enabled: true
```

Setting this property configures Tomcat to dispatch incoming HTTP requests to an ephemeral virtual thread created by `Executors.newVirtualThreadPerTaskExecutor()`.

1.2.1 The Virtual Thread Bottleneck Fallacy

The effortless activation of virtual threads catalyzed a widespread industry misconception that this dissertation defines as **The Virtual Thread Bottleneck Fallacy**. This fallacy presumes that the primary limiting factor in enterprise backend performance is always web-tier container thread pool concurrency, and that enabling virtual threads universally eliminates concurrency bottlenecks, drastically cuts latencies, and delivers immediate order-of-magnitude throughput gains.

In reality, removing the platform thread ceiling does not eliminate performance bottlenecks; rather, it shifts the operational pressure points deeper into the application architecture:

1. **Domain Serialization Ceiling:** Enterprise workflows frequently contain shared mutable state protected by mutual exclusion (such as Java intrinsic monitors, explicit locks, or database table locks). By Amdahl's Law [1], if an operation contains a serial duration D, the maximum system service rate is fundamentally bounded by $\mu = 1/D$, irrespective of whether requests are served by platform threads or virtual threads.

2. **Carrier Pinning Hazards:** In Java 21, when a virtual thread blocks inside an intrinsic monitor (`synchronized` block or method), the JVM runtime cannot unmount the virtual thread from its carrier thread. This phenomenon—*monitor pinning*—locks the underlying OS platform thread in place. Under concurrent load, carrier threads are rapidly exhausted, starving the entire scheduler.

3. **Silent Native Carrier Captivity:** When a virtual thread invokes native code via the Java Native Interface (JNI), blocking POSIX system calls (e.g., `nanosleep()`) execute entirely outside HotSpot continuation management. The OS carrier thread is captured in an unmanaged wait state, starving the virtual thread pool without generating any Java Flight Recorder (JFR) pinning telemetry.

4. **Downstream Connection Pool Exhaustion:** In relational persistence, holding a database connection across a blocking operation shifts queueing entirely to the JDBC connection pool. Even though virtual threads unmount cleanly without pinning carrier threads, total throughput is capped by pool capacity ($X_{\max} = K/T_{\text{hold}}$), causing queue formation at the database pool checkout tier.

1.3 Research Goals, Questions, and Hypotheses

To establish a rigorous, empirical systems investigation into these phenomena, this dissertation addresses two canonical Research Questions:

Research Question 1 (RQ1):
How does synchronization (specifically monitor pinning) interact with carrier-thread availability, application-level serialization, and runtime throughput in Java 21 Spring Boot MVC systems?

Research Question 2 (RQ2):
What architectural risks—including native boundaries, connection pool management, context propagation, and observability limits—must engineering teams evaluate before enabling Virtual Threads in existing Spring Boot MVC systems?

To ensure answerability and rigorous empirical evaluation, Table 1.1 operationalizes each research question into concrete treatments, hypotheses, and required evidentiary criteria.

Table 1.1: Research questions, operational deconstruction, and intended evidence.

ID	Operational Focus	Treatment & Variables	Intended Evidentiary Criteria
RQ1	Carrier pinning vs. domain serialization ceiling	Factorial matrix: Tomcat platform vs. virtual threads; `synchronized` vs. `ReentrantLock` vs. no-lock; Java 21 LTS vs. Java 24 (JEP 491).	JFR `jdk.VirtualThreadPinned` events (108 to 0); steady-state throughput (1.956 vs. 1.967 req/s); Amdahl ceiling proof ($\mu \approx 2.0$ req/s); AQS barging dispersion.
RQ2	Architectural risks: JNI, pools, context, diagnostics	POSIX JNI C downcalls; HikariCP connection holding (Stratum 8 two-treatment design: pool size 1 vs. 10); `ThreadLocal` lifecycle; JFR threshold limits.	Process CPU ($1.04\% \pm 0.12\%$) in Darwin/Mach wait state with 0 JFR pinning events; connection queue growth (49 queued threads); memory leak traces; diagnostic blind spot analysis.

To prevent conflating distinct performance failure modes, Table 1.2 defines an authoritative *Symptom-to-Mechanism Disambiguation Matrix*. Every observed performance anomaly must satisfy its explicit evidentiary requirement before assigning architectural causality.

Table 1.2: Symptom-to-Mechanism Disambiguation Matrix.

Observed Symptom	**Naive Hypothesis**	**True Underlying Mechanism**	**Evidentiary Requirement**
Throughput plateau at ≈ 2 req/s	Virtual threads fail to unmount due to monitor pinning	**Domain Serialization:** Critical section (500 ms) caps service rate to $\mu = 1/D = 2.0$ req/s via Amdahl's Law.	Throughput identical (1.956 vs. 1.967 req/s) when pinning is eliminated via `ReentrantLock`.
Median latency inflation ($R \approx 5.0$ s)	Scheduler inefficiency or context-switching storm	**Closed-Loop Saturation:** 10 Virtual Users (VUs) with $Z = 100$ ms think time: Little's Law gives $R = N/X - Z \approx 4.985$ s.	Mathematical convergence between Little's Law formula and k6 steady-state response distribution.
Throughput collapse with zero JFR pinning events	Virtual threads unmounting cleanly; network I/O fault	**Silent JNI Carrier Captivity:** POSIX `nanosleep()` holds carrier in `_thread_in_native`; HotSpot yield bypassed.	k6 throughput drops to 1.97 req/s while JFR pinning event count is strictly 0.
Carrier starvation with process CPU $\approx 1\%$	Carrier threads in 100% busy-spin loop	**Kernel Wait Suspension:** Native POSIX sleep places thread in Darwin/Mach wait state without burning CPU.	Process telemetry showing steady-state CPU of $1.04\% \pm 0.12\%$.
Connection pool starvation without pinning	Virtual threads eliminate pool sizing limits	**Downstream Bottleneck Shift:** Clean unmounting (0 JFR events), but throughput capped by pool ($X_{\max} = K/T_{\text{hold}}$).	HikariCP telemetry (1.0/1 active, 49 queued VUs) and scaling (3.27 to 32.60 req/s) with 0 pinning.
Tail latency dispersion in refactored locks	`ReentrantLock` is slower than intrinsic monitor	**AQS NonfairSync Barging:** Default non-fair lock permits arriving threads to bypass queue, inflating variance.	Latency standard deviation spread ($\sigma = 16.53$ ms for `ReentrantLock` vs. 0.70 ms for `synchronized`).

1.4 Scope, Assumptions, Constraints, and Exclusions

To maintain rigorous experimental control and internal validity, the scope and boundaries of this study are defined as follows:

- **Software Stack:** The primary research artifact is an instrumented enterprise benchmark application implemented on **Spring Boot 3.5.12**, executed on embedded **Apache Tomcat 10.1.52**, and compiled with **Java 21 LTS** (Azul Zulu 21.0.8, OpenJDK HotSpot 64-bit Server VM). Comparative boundary replications utilize early-access OpenJDK 24.0.2 (build 24+36).
- **Persistence Layer:** Database access is modeled using Spring `JdbcTemplate` backed by **HikariCP 6.3.3**, evaluated across in-memory H2 databases and containerized Docker PostgreSQL 15-alpine instances.
- **Hardware and Operating System Environment:** All primary benchmark trials were executed on a dedicated, bare-metal Apple Silicon host:
  - **Processor:** Apple M1 (8 logical cores: 4 performance cores, 4 efficiency cores).
  - **Memory:** 8 GB unified memory.
  - **Operating System:** Darwin 25.2.0 (macOS 15.2, arm64).
- **Scheduler Isolation Parameterization:** To isolate carrier starvation mechanics with mathematical clarity and prevent multi-core scheduling jitter from masking single-carrier contention, primary experimental trials applied JVM scheduler flags:

  ```
  -Djdk.virtualThreadScheduler.parallelism=1
  -Djdk.virtualThreadScheduler.maxPoolSize=1
  ```

  This constitutes an in-process JVM scheduler constraint rather than an OS cgroup quota, guaranteeing that the carrier pool consists of exactly one OS platform thread.
- **Closed-Loop Workload Model:** Load generation was conducted using Grafana k6 [17] in a closed-loop configuration (fixed concurrency with constant think time $Z = 100$ ms), enforcing stationary queueing equilibrium and preventing client-side packet drops.

1.4.1 Explicit Methodological Exclusions

The following domains are explicitly excluded from the empirical scope:

1. **Reactive Frameworks as Empirical Treatments:** Monadic reactive stacks (Spring WebFlux, Project Reactor) are compared theoretically, but are not benchmarked as empirical treatments.

2. **Distributed Multi-Node Topologies:** Clustered network meshes and cross-datacenter latencies are excluded to eliminate non-deterministic network noise from JVM scheduler measurements.

3. **Inferential Hypothesis Testing on Small Samples:** Primary benchmark conditions were executed across $N = 3$ independent process-level repetitions. These repetitions are reported strictly as descriptive engineering bounds; inferential testing (p-values, ANOVA) is excluded.

4. **Flawed Exploratory Runs:** Preliminary exploratory runs lacking proper JFR configuration or connection-holding controls are formally partitioned into Stratum 7 and excluded from quantitative findings.

1.5 Contributions and Responsibility Boundaries

This dissertation delivers four central scientific and architectural contributions, supported by the ratified empirical evidence corpus:

Table 1.3: Contribution and responsibility boundary ledger.

ID	Original substance	Evidence later in thesis	Responsibility boundary
C1	Decoupling carrier unpinning from domain serialization	Chapter 6 (§6.2, Stratum 1: 12 trials, ΔTP $< 0.3\%$)	Student independent empirical discovery, measurement, and theoretical derivation.
C2	Silent carrier starvation via native JNI POSIX downcalls	Chapter 6 (§6.3, Stratum 3: 27 trials, 0 JFR pinning events)	Student independent native C library implementation, telemetry design, and verification.
C3	Dual-window reporting standard and 125-run corpus audit	Chapters 4 and 6 (504 artifacts, SHA-256 verifications)	Student independent execution, open-science freeze curation, and Python validation scripts.
C4	Authoritative engineering migration framework	Chapter 7 (§7.3: 5 rules, 4 refactorings, decision tree)	Student independent synthesis based directly on empirical testbed findings.

The student's independent contributions and responsibility boundaries (summarized in Table 1.3) reflect the following achievements:

- **C1 (Carrier Unpinning vs. Serialization):** Demonstrating through empirical benchmarking and queueing theory that refactoring `synchronized` to `ReentrantLock` completely eliminates carrier pinning (108 to 0 events), but yields identical throughput (1.956 vs. 1.967 req/s) due to the Amdahl serialization ceiling ($\mu \approx 2.0$ req/s).

- **C2 (Silent JNI Carrier Starvation):** Proving that blocking native POSIX system calls hold carrier threads captive in Darwin/Mach wait states with low process CPU ($1.04\% \pm 0.12\%$) and exactly zero JFR pinning events, exposing a critical observability gap.

- **C3 (Dual-Window Reporting Standard):** Establishing an open-science corpus of 125 physical trials across 8 strata with 504 cataloged artifacts (966 MB) and SHA-256 integrity verification, formulating the Dual-Window Reporting Standard to filter 25.8% ramp-up latency distortion.

- **C4 (Authoritative Migration Framework):** Providing an evidence-backed migration framework comprising 5 core architectural rules, 4 correctness-preserving refactoring patterns, and a deterministic decision tree.

1.5.1 The Three Foundational Architectural Principles

Synthesizing these findings yields three foundational architectural principles governing virtual thread concurrency:

Principle 1: The Amdahl-Little Invariant (Capacity Conservation).
Virtual threads expand in-flight concurrency (N), but cannot increase the service capacity ($\mu = 1/D$) of serialized application critical sections or downstream resources. When an architectural bottleneck dominates request duration, expanding concurrency merely inflates queueing latency ($R = N/X - Z$) in accordance with Little's Law, yielding zero throughput benefit.

Principle 2: The Boundary Invariant (Diagnostic and OS Transparency).
Cooperative carrier unmounting operates strictly within managed JVM boundaries. When a virtual thread crosses into unmanaged native execution (JNI / POSIX syscalls), it captures the physical OS carrier thread. Because native blocking bypasses HotSpot continuation yield points, carrier starvation occurs silently—evading standard JFR pinning diagnostics and maintaining negligible CPU utilization.

Principle 3: The State and Resource Invariant (Thread Ephemerality).
Virtual threads transform threads from heavyweight, pooled resources into ephemeral execution units. Consequently, architectural patterns designed around thread pooling—such as using thread pools to throttle downstream database access or caching expensive state in `ThreadLocal` variables—become severe anti-patterns. Concurrency bounding must be explicitly migrated to resource-specific semaphores and connection pools, while request context must enforce deterministic teardown.

1.6 Thesis Roadmap

The remainder of this dissertation is organized as follows:

- **Chapter 2 (Related Work):** Synthesizes foundational concurrency literature, formalizes a five-domain taxonomy, and establishes a 10-point Comparative Delta Matrix against closest related work.

- **Chapter 3 (Background and Foundations):** Details JVM concurrency evolution, Project Loom internals, HotSpot continuation freeze mechanics, JEP 491 unpinning, and queueing foundations (Little's Law and Amdahl's Law).
- **Chapter 4 (Research Design and Methodology):** Formulates the empirical systems research paradigm, the 8-strata reference framework, the closed-loop k6 harness, JFR telemetry, and the Dual-Window Reporting Standard.
- **Chapter 5 (Implementation and Research Infrastructure):** Documents the Spring Boot 3.5.12 testbed architecture, POSIX JNI C shared library, HikariCP configuration, and automated test suite.
- **Chapter 6 (Experimental Results):** Reports empirical measurements across all 125 benchmark trials, presenting side-by-side whole-run and steady-state metrics, JFR event counts, and AQS dispersion signatures.
- **Chapter 7 (Analysis, Discussion, and Migration Guidance):** Provides formal closure for RQ1 and RQ2, details the Three-Tier Concurrency Taxonomy, master threats to validity, and delivers the authoritative migration framework.
- **Chapter 8 (Conclusion):** Summarizes key research findings, limitations, lessons learned, and outlines five future research directions.
- **Appendices A–C:** Detail reproducibility procedures, research ethics and AI tool declarations, and the master process checklist.

Chapter 2

Related Work

2.1 Review Scope and Source Strategy

To establish the academic and engineering context of this research, a systematic literature review was conducted across five foundational concurrency domains:

1. **Queueing and Scheduling Foundations:** Classical single- and multi-server queueing networks, operational conservation laws, and the historical evolution of two-level ($M : N$) user-space thread scheduling.

2. **Concurrency, Memory Models, and Synchronization:** Language-level intrinsic monitors, hardware lock designs, explicit lock synchronization, and the Java Memory Model (JMM).

3. **Empirical Benchmarking Methodology in Managed Runtimes:** Rigorous protocols for warm-up isolation, steady-state identification, non-independent error handling, and closed-loop workload modeling.

4. **Normative JDK Specifications and OpenJDK Architecture:** Platform specifications defining Virtual Threads, carrier thread multiplexing, continuation capture, and monitor unpinning proposals.

5. **Empirical Evaluations of Modern Asynchronous and Virtual-Thread Frameworks:** Contemporary benchmark studies evaluating Project Loom, Spring Boot, Quarkus, and real-world enterprise adoption post-mortems.

The search strategy targeted peer-reviewed computer science literature published between 1961 and 2026 across IEEE Xplore, the ACM Digital Library, SpringerLink, and

ScienceDirect, supplemented by normative OpenJDK Enhancement Proposals (JEPs) [18, 40, 41], Java Community Process standards (JSR-133 [31]), and industrial production post-mortems [11]. Works were selected based on direct relevance to thread multiplexing, synchronization bottlenecks, or managed runtime benchmarking.

2.2 Five-Domain Concurrency Taxonomy

Table 2.1 categorizes the foundational and empirical literature across the five identified domains, detailing the core focus of each source and its evidentiary boundaries relative to this dissertation.

Table 2.1: Comprehensive literature taxonomy across foundational concurrency domains.

Domain	**Source**	**Primary Scientific Value**	**Evidentiary Boundary**
Queueing & Scheduling	Amdahl (1967) [1]	Formulates speedup ceiling governed by strictly serial fraction $(1 - P)$.	Theoretical scaling model; predates user-space thread unmounting.
Queueing & Scheduling	Kleinrock (1975) [22]	Mathematical foundations of $M/M/1$ and $M/G/1$ queueing systems.	Theoretical queueing theory; does not evaluate JVM carrier pools.
Queueing & Scheduling	Little (1961, 2011) [29, 30]	Operational queueing law ($L = \lambda W$) relating population, throughput, and latency.	Steady-state invariant; does not model internal lock contention.
Queueing & Scheduling	Anderson et al. (1991) [2]	Establishes $M : N$ two-level scheduling, user unmounting, and scheduler activations.	OS kernel implementation; predates managed runtimes and JDBC pools.

Domain	Source	Primary Scientific Value	Evidentiary Boundary
Queueing & Scheduling	Dijkstra (1965) [10]	Foundational formulation of mutual exclusion and serialized execution regions.	Algorithmic primitive; does not address carrier thread starvation.
Queueing & Scheduling	Hoare (1974) [20]	Conceptual origin of structured monitors and condition wait queues.	Language design concept; does not foresee carrier pinning hazards.
Queueing & Scheduling	Ousterhout (1996) [37]	Argued against thread synchronization and memory overhead in favor of events.	Systems debate; predates modern lightweight user-mode thread multiplexers.
Queueing & Scheduling	Welsh et al. (2001) [47]	Staged Event-Driven Architecture (SEDA) decomposing systems into event queues.	Event-driven pipeline; introduces asynchronous programming complexity.
Queueing & Scheduling	von Behren et al. (2003) [46]	Showed high-concurrency servers scale using lightweight threads and compiler support.	Early user-threading rationale; predates Java 21 continuations.
Concurrency Models	Lamport (1979) [23]	Formalized sequential consistency for multiprocessor systems.	Hardware memory model; does not evaluate thread scheduling.

Domain	Source	Primary Scientific Value	Evidentiary Boundary
Concurrency Models	Herlihy & Shavit (2008) [19]	Mathematical foundations of mutual exclusion and queue locks.	Focuses on multi-core hardware; does not model carrier pools.
Concurrency Models	Goetz et al. (2006) [15]	Canonical reference on `java.util.concurrent` and explicit locking.	Written for 1:1 platform threads; predates Loom unmounting.
Concurrency Models	Lea (2000) [25, 26]	Engineered AQS and work-stealing ForkJoinPool powering Loom scheduler.	Runtime foundation; does not evaluate full Spring MVC request lifecycles.
Concurrency Models	Manson et al. (2005) [31]	Formulated Java Memory Model (JSR-133) happens-before consistency.	Memory visibility semantics; does not analyze throughput under load.
Concurrency Models	Mellor-Crummey & Scott (1991) [32]	Introduced MCS FIFO queue locks, foundation of HotSpot AQS.	Hardware algorithms; predates Java virtual thread parking.
Concurrency Models	Bacon et al. (2001) [4]	Optimized Java monitor acquisition via mark-word CAS fast paths (thin locks).	JVM monitor design; does not address carrier pinning in monitors.
Benchmarking Method	Georges et al. (2007) [14]	Rigorous principles for warm-up, steady-state, and multi-process variance.	Evaluates JIT/GC steady-state; does not evaluate virtual thread harnesses.

Domain	Source	Primary Scientific Value	Evidentiary Boundary
Benchmarking Method	Blackburn et al. (2006) [6]	DaCapo benchmark suite standardizing macro-level Java evaluation.	Macro-benchmarks predating virtual threads; no web I/O evaluation.
Benchmarking Method	Kalibera & Jones (2013) [21]	Treatment of correlated measurement errors and process repetitions.	Methodological foundation; requires independent process invocations.
Benchmarking Method	Bulej et al. (2017) [7]	Framework for automated performance regression detection in CI pipelines.	Regression testing; provides discipline for reproducible suites.
Benchmarking Method	Traini et al. (2022) [44]	Dual-window standard separating warm-up ramp transients from steady state.	Methodological literature; does not evaluate Virtual Threads.
Benchmarking Method	Schroeder et al. (2006) [43]	Proved closed-loop harnesses enforce self-throttling; open-loop tests capacity.	Justifies closed-loop k6 testing to isolate internal mechanisms.
JDK Specifications	JEP 425 & 436 [38, 39]	Preview releases of Virtual Threads in JDK 19 and JDK 20.	Incubation specifications; superseded by Java 21 LTS standard.

Domain	Source	Primary Scientific Value	Evidentiary Boundary
JDK Specifications	Pressler (JEP 444, 2023) [40]	Normative specification of Virtual Threads and monitor pinning in Java 21 LTS.	Platform specification; does not quantify application lock interactions.
JDK Specifications	Pressler (JEP 491, 2024) [41]	Redesigns HotSpot monitors to allow unmounting inside `synchronized`.	Platform enhancement; provides no empirical data for Spring MVC.
JDK Specifications	JEP 506 (2025) [18]	Scoped Values as immutable context sharing to replace `ThreadLocal`.	Future feature; does not resolve `ThreadLocal` in Java 21 systems.
JDK Specifications	Lindholm et al. (JVMS 21) [28]	Normative JVM bytecode instructions, monitors, and thread states.	Specification; does not evaluate application throughput under load.
JDK Specifications	Gosling et al. (JLS 21) [16]	Normative Java language semantics, monitors, and memory model (§17).	Language specification; does not model carrier thread multiplexing.
JDK Specifications	Oracle (2023) [36]	Official guidance and best practices for virtual thread adoption.	General guidance; provides no empirical data under carrier limits.

Domain	Source	Primary Scientific Value	Evidentiary Boundary
Modern Frameworks	Beronić et al. (2021) [5]	Early exploratory analysis of structured concurrency in Loom prototypes.	Evaluated pre-standard Loom; does not evaluate Java 21 LTS or JFR.
Modern Frameworks	Navarro et al. (2023) [34]	Quarkus benchmark showing reactive core outperformed VT near saturation.	Focuses on Quarkus reactive; does not isolate locks, JDBC, or JNI.
Modern Frameworks	Lašić et al. (2024) [24]	Evaluated aggregate virtual thread throughput against database backends.	Evaluated gross client metrics; omitted connection-holding control, JFR unmounting verification, and JNI captivity.
Modern Frameworks	Likus & Krużel (2025) [27]	Evaluated I/O vs. CPU REST endpoints on Spring Boot with Java 21.	Relies on black-box HTTP metrics; omitted lock isolation, JFR stack traces, JNI downcalls, and warm-up filtering.
Modern Frameworks	Charlak et al. (2026) [8]	Comparative web server study showing reactive memory advantages.	Focuses on paradigm comparison; lacks migration refactoring guidance.

Domain	Source	Primary Scientific Value	Evidentiary Boundary
Modern Frameworks	dotCMS Core (2024) [11]	Industrial rollback: reverted async cache commits due to lock pinning.	Industry post-mortem; confirms failure modes but lacks controlled benchmark data.

2.3 Closest Prior Work and Comparative Delta Matrix

Among contemporary empirical evaluations of Java 21 Virtual Threads, two studies represent the closest prior work to this dissertation:

1. **Likus & Krużel (2025) [27]:** Investigated the throughput and latency of synthetic I/O-bound (using non-blocking sleep) and CPU-bound endpoints on Spring Boot 3 using Java 21. While demonstrating high concurrency under unconstrained simulated I/O, the study treated the JVM as a black box: it measured only HTTP client-side metrics, evaluated no critical sections protected by mutual exclusion, performed no internal JVM Flight Recorder telemetry, and omitted foreign-function and persistence boundaries.

2. **Lašić et al. (2024) [24]:** Evaluated virtual threads against relational database backends, observing that connection pool exhaustion caps scalability. However, their experimental design evaluated only gross aggregate client throughput without isolating connection checkout duration (T_{hold}), omitted mathematical verification of queueing conservation laws ($X_{\max} = K/T_{\text{hold}}$), did not verify carrier unmounting via JFR traces, and omitted foreign function (JNI) boundaries entirely.

To establish the rigorous scientific differentiation of this research, Table 2.2 provides a 10-point Comparative Delta Matrix contrasting this dissertation with Likus & Krużel (2025) [27] and Lašić et al. (2024) [24].

Table 2.2: Systematic Comparative Delta Matrix against closest related work.

Dimension	**Likus & Krużel (2025) [27]**	**Lašić et al. (2024) [24]**	**This Dissertation**
Primary Framework	Spring Boot 3.x	Java 21 Server Apps	Spring Boot 3.5.12 MVC (Zulu 21.0.8)
JVM Telemetry	None (client HTTP only)	None (client metrics only)	Continuous JFR stack traces (`jdk.VirtualThreadPinned`, 20 ms)
Domain Mutex Isolation	None (pure I/O sleep)	None (uncontended queries)	Strict factorial matrix: `synchronized` vs. `ReentrantLock` vs. no-lock under 500 ms critical section
Carrier Pinning Delta	Not measured	Not measured	Measured: 108 pinning events (`synchronized`) reduced to **0 events** (`ReentrantLock`)
Amdahl Invariant Proof	Not investigated	Not investigated	Formally demonstrated: eliminating pinning yields **identical throughput** (1.956 vs. 1.967 req/s)
Foreign Functions (JNI)	Not evaluated	Not evaluated	Proven: blocking POSIX `nanosleep()` silently captures carrier (1.04% CPU, 0 JFR events)
Database Pool Model	Not evaluated	Aggregate pool throughput	Factorial ablation (Stratum 8): connection-held vs. unheld delay ($X_{\max} = K/T_{\text{hold}}$)
Queueing Calibration	None	Informal observations	Exact mathematical validation of Little's Law ($R = N/X - Z$)
Steady-State Isolation	Whole-run client averages	Whole-run averages	Dual-window reporting standard (55 s whole-run vs. 30 s steady-state hold)
Migration Framework	High-level observations	General pool suggestions	5 Mandatory Rules, 4 Production Refactorings, Master Decision Tree

2.4 Critical Synthesis and the Research Gap

Synthesizing this literature reveals a coherent theoretical and methodological foundation:

1. **Conservation Laws Transcend Scheduling Models:** As established by Amdahl [1], Little [29, 30], and Kleinrock [22], user-mode thread multiplexing cannot overcome queueing bottlenecks. When an application contains a serialized critical section or interacts with a bounded downstream database pool, thread unmounting merely transfers queue accumulation across architectural tiers.

2. **Two-Level Scheduling and Boundary Hazards:** The historical threading debate (Ousterhout [37] vs. Welsh et al. [47] vs. von Behren et al. [46]) highlighted the cognitive clarity of imperative threading against event-loop scalability. Anderson et al. [2] showed that $M : N$ threading requires kernel-user cooperative signaling. In Java 21 Virtual Threads (JEP 444 [40]), unmounting relies on runtime cooperation via continuations and AQS [26]; however, foreign function invocations (JNI) bypass runtime cooperation, reproducing the exact carrier captivity problems identified in early operating systems research.

3. **Rigorous Managed-Runtime Benchmarking:** Methodologies by Georges et al. [14], Kalibera & Jones [21], and Traini et al. [44] mandate that JVM performance evaluations must rigorously decouple transient warm-up phases from steady-state equilibrium, treat independent process executions as units of variance, and avoid unjustified inferential claims on small sample sizes.

4. **Transitory Nature of Java 21 Synchronization Constraints:** While explicit locks (`ReentrantLock`) [15, 19] resolve monitor pinning in Java 21, JEP 491 [41] demonstrates that monitor pinning is a temporary runtime limitation destined for removal in JDK 24. Migration guidance must therefore separate transient JDK runtime behavior from permanent architectural bottlenecks.

2.4.1 The Identified Research Gap

Despite the breadth of prior literature, a critical **research gap** exists in the systems engineering body of knowledge:

Prior empirical studies have failed to provide a controlled, reproducible, end-to-end Spring Boot MVC investigation that jointly isolates:

1. A Java 21 `synchronized` monitor-pinning treatment alongside an identical `ReentrantLock` refactored treatment under an unbroken business endpoint contract;
2. Controlled platform thread and diagnostic baseline comparisons on the identical hardware artifact;
3. Continuous, low-overhead Java Flight Recorder (JFR) stack attribution verifying the exact duration and method origin of `jdk.VirtualThreadPinned` events;
4. A clear empirical decoupling of carrier thread unmounting from application-level mutual exclusion serialization (Amdahl's Law);
5. An empirical deconstruction of silent carrier captivity across foreign function (JNI) boundaries, distinguishing OS kernel wait-states from active CPU utilization; and
6. A two-treatment factorial ablation of relational persistence, validating the Downstream Bottleneck Shift Law under HikariCP connection holding.

2.5 Positioning of the Proposed Contribution

This dissertation closes this research gap by delivering an empirical systems investigation grounded in physical benchmark data. Rather than proposing speculative conceptual architectures or relying on black-box HTTP client metrics, this work constructs an instrumented enterprise Spring Boot 3.5.12 testbed, subjects it to 125 audited benchmark trials across 8 experimental strata, and combines low-level HotSpot C++ source inspection with JFR telemetry and OS process monitoring. The resulting contributions provide both theoretical validation of queueing conservation laws and actionable engineering guidance for practitioners migrating enterprise Java backend systems to Virtual Threads.

Chapter 3

Background and Foundations

3.1 Evolution of Concurrency in the Java Virtual Machine

The architectural history of concurrency in the Java Virtual Machine (JVM) reflects an ongoing tension between programming ergonomics, memory efficiency, and operating system resource management:

1. **Green Threads (JDK 1.1):** Early JVM implementations multiplexed user-mode threads onto a single operating system process in user space ($N : 1$ model). While lightweight, green threads could not exploit multi-core Symmetric Multiprocessing (SMP) architectures and suffered from coarse-grained scheduling: any blocking system call issued by a single green thread halted the entire JVM process.

2. **Native Platform Threads (JDK 1.2 to Java 20):** To harness multi-core hardware, OpenJDK HotSpot transitioned to a strict $1 : 1$ threading model, delegating thread management to the host operating system kernel (via POSIX `pthread`s on Linux and macOS, or Win32 threads on Windows). Each instance of `java.lang.Thread` mapped directly to an OS kernel thread. While this unlocked true hardware parallelism, it tethered Java concurrency directly to heavyweight OS kernel abstractions:

    - Fixed call-stack virtual memory reservations (typically 1 024 KB via `-Xss1m`).
    - Kernel-space context switching overhead ($ring\ 3 \leftrightarrow ring\ 0$ mode transitions and CPU cache invalidations).

- The necessity of bounded thread pooling (e.g., Tomcat `maxThreads = 200`) to prevent memory exhaustion.

3. **The Reactive Interregnum (2010s):** As web traffic expanded, non-blocking, asynchronous reactive frameworks (e.g., Netty, Spring WebFlux, Project Reactor) emerged to multiplex thousands of concurrent connections over a small pool of event-loop threads matching CPU core counts. While highly scalable, reactive programming imposed severe cognitive friction (monadic composition via `Mono` and `Flux`), broken stack traces, and deep architectural friction when interfacing with blocking relational database drivers (JDBC).

4. **Project Loom and Virtual Threads (Java 21 LTS, JEP 444 [40]):** Project Loom resolves this dichotomy by decoupling the Java thread abstraction from the OS kernel thread, introducing lightweight, user-mode *Virtual Threads* managed directly by the JVM runtime under an $M : N$ scheduling model.

3.2 Project Loom and Virtual Thread Architecture

Under JEP 444 [40], Virtual Threads operate as user-mode tasks scheduled by the JVM onto a bounded pool of standard OS platform threads, designated as *carrier threads*.

3.2.1 Continuation Abstraction and Chunked Stack Management

At the core of Project Loom is the `jdk.internal.vm.Continuation` abstraction. A continuation represents a sequential computation that can suspend its execution voluntarily (yield) and resume later from the exact suspension point.

Unlike platform threads, which allocate a fixed, contiguous OS memory stack, a virtual thread maintains its execution stack dynamically on the garbage-collected Java heap. When a virtual thread is instantiated, its initial memory footprint is small ($\approx$ 1 KB). Stack frames are allocated as a linked chain of *stack chunks* directly on the heap:

- **Stack Mounting:** When scheduled to run, the JVM *mounts* the virtual thread onto a carrier thread. The continuation's top execution frames are copied from the heap onto the native OS execution stack of the carrier thread.

- **Stack Unmounting (Yielding):** When the virtual thread encounters a blocking operation (such as socket read/write, explicit lock acquisition, or sleep), the JVM

captures and freezes the execution frames from the carrier thread's native stack back into heap-allocated stack chunks. The virtual thread *unmounts*, freeing the carrier thread to execute other ready virtual threads.

- **Stack Remounting:** Once the blocking operation completes (e.g., the OS signals socket readiness via `epoll` or `kqueue`), the virtual thread is unparked and submitted to the carrier pool's scheduling queue, where an available carrier thread remounts it and resumes execution.

3.2.2 The Work-Stealing Carrier Scheduler

The default carrier scheduler for virtual threads is an instance of `java.util.concurrent.ForkJoinPool`, initialized with work-stealing semantics [26]:

- **Parallelism Level:** Governed by JVM property `jdk.virtualThreadScheduler.parallelism`, defaulting to the available core count ($\min(\text{cores}, 256)$).
- **Maximum Carrier Pool Size:** Governed by JVM property `jdk.virtualThreadScheduler.maxPoolSize`, defaulting to 256.
- **Carrier Identification:** Carrier threads are standard platform daemon threads named `ForkJoinPool-1-worker-N`.

3.3 Mechanics of Monitor Pinning in Java 21

In Java 21, the cooperative unmounting mechanism is subject to a critical runtime limitation known as **carrier pinning**. When a virtual thread is pinned, it cannot unmount from its carrier thread upon encountering a blocking operation. Consequently, the underlying OS carrier thread remains blocked, unavailable to schedule any other virtual thread.

3.3.1 HotSpot C++ Continuation Freeze Mechanics

In the OpenJDK HotSpot C++ source code (`src/hotspot/share/runtime/continuationFreeze.cpp`) [35], continuation suspension is implemented by the `freeze()`

function. When a virtual thread attempts to yield, the runtime walks the call stack from the current frame to the continuation delimiter.

During stack-walking, HotSpot evaluates whether the thread holds any intrinsic object monitors:

```
// HotSpot C++ continuationFreeze.cpp snippet
if (thread->held_monitor_count() > 0) {
    return freeze_pinned_monitor;
}
```

If the virtual thread has entered an intrinsic monitor via the `monitorenter` bytecode instruction (i.e., executing inside a `synchronized` method or block) and executes a blocking operation, the freeze state machine returns `freeze_pinned_monitor`. The JVM aborts the unmounting sequence, and the carrier thread transitions to a blocked OS state.

HotSpot classifies pinning under two primary conditions:

1. **Monitor Pinning:** Holding an intrinsic monitor (`synchronized`) during a blocking call.
2. **Native Frame Pinning:** Having native C/C++ stack frames on the call stack (e.g., executing inside a JNI downcall or foreign function invocation).

3.3.2 The JDK Runtime Evolution: JEP 491 in Java 24

To address monitor pinning, OpenJDK introduced **JEP 491** (*Synchronize Virtual Threads without Pinning*) [41], integrated into JDK 24. JEP 491 redesigns HotSpot's `ObjectMonitor` implementation. Instead of associating monitor acquisition with native thread structures, the runtime permits virtual threads to release the carrier thread even when blocking inside `synchronized` blocks or waiting on monitor condition variables (`Object.wait()`).

However, JEP 491 explicitly does **not** eliminate all forms of carrier pinning:

- **Native Frames Remain Pinned:** Blocking calls executed while native frames exist on the stack continue to pin the carrier thread.
- **Class Initializer Execution:** Code executing inside static class initializers (`<clinit>`) remains pinned.

- **Domain Serialization Remains Unchanged:** JEP 491 resolves the *schedulability* bottleneck (carrier availability), but does not alter *mutual exclusion*. A critical section that serializes execution continues to cap throughput to $\mu = 1/D$, as governed by Amdahl's Law [1].

3.4 Synchronization Primitives: Intrinsic Monitors vs. Explicit Locks

In Java, mutual exclusion can be achieved through two distinct mechanisms with contrasting runtime implementations:

3.4.1 Language-Level Intrinsic Monitors (`synchronized`)

Intrinsic monitors are built directly into the Java language and bytecode specification [16, 28]. Every Java object possesses an intrinsic monitor represented in the object's mark word. In HotSpot, monitor acquisition uses biased locking (historically) and thin locks optimized via atomic compare-and-swap (CAS) operations [4]. When contention occurs, the thin lock is inflated into a heavyweight C++ `ObjectMonitor` containing OS-level mutexes and wait queues. In Java 21, because the inflated `ObjectMonitor` maintains direct pointers to the calling OS thread, virtual threads cannot yield while holding it.

3.4.2 Explicit Locks and the AbstractQueuedSynchronizer (AQS)

The `java.util.concurrent.locks` framework, engineered by Doug Lea [25], implements synchronization in user space via the `AbstractQueuedSynchronizer` (AQS) [26], which is derived from the MCS queue lock algorithm [32].

In `ReentrantLock`, thread parking and unparking are managed through `LockSupport.park()` and `LockSupport.unpark()`. When a virtual thread encounters contention on a `ReentrantLock`:

1. The virtual thread enqueues an AQS node representing its wait state.
2. `LockSupport.park()` invokes the JVM continuation yield hook.
3. The virtual thread unmounts cleanly from its carrier thread, avoiding monitor pinning entirely.

3.4.3 The NonfairSync Barging Anomaly

By default, `ReentrantLock` uses a non-fair synchronization policy (`NonfairSync`). Unlike fair locks, which strictly respect FIFO queue ordering, `NonfairSync` allows newly arriving threads to execute a direct CAS acquisition on the lock state before inspecting the AQS wait queue. This "barging" behavior maximizes raw throughput by avoiding thread unpark overhead. However, under high virtual thread concurrency, barging induces substantial variance in wait times, producing elevated tail-latency dispersion (σ) compared to intrinsic monitors.

3.5 Persistence Concurrency: JDBC and Connection Pools

Enterprise Spring Boot MVC applications typically interact with relational databases through the Java Database Connectivity (JDBC) API. Standard JDBC drivers (e.g., PostgreSQL JDBC, MySQL Connector/J) execute synchronous, blocking network I/O over standard TCP sockets.

3.5.1 HikariCP Connection Pool Mechanics

Because creating physical database connections is computationally expensive (involving TCP handshakes, TLS negotiation, and database backend process spawning), enterprise applications maintain bounded connection pools via HikariCP [49].

HikariCP regulates access to a fixed pool of K database connections using a specialized concurrent data structure (`FastList` and `ConcurrentBag`). When an incoming request executes a database query:

1. The thread requests a connection via `DataSource.getConnection()`.
2. If an idle connection is available, it is leased immediately.
3. If all K connections are active, the calling thread blocks on a synchronous handoff queue (`SynchronousQueue`) until a connection is returned or a timeout (`connectionTimeout`, default 30 s) expires.

3.5.2 The Downstream Bottleneck Shift Law

When virtual threads execute JDBC queries, standard socket blocking within the JDBC driver unmounts cleanly from carrier threads (provided the driver does not use `synchronized` blocks). However, virtual threads cannot expand downstream database capacity.

If an application holds a leased database connection across a blocking operation of duration T_{hold}, the maximum theoretical system throughput $X_{\max}$ is strictly constrained by the connection pool size K and Little's Law [29]:

$$X_{\max} = \frac{K}{T_{\text{hold}}} \tag{3.1}$$

Multiplexing 10,000 virtual threads onto the application tier cannot exceed this ceiling; instead, concurrent requests accumulate in the HikariCP checkout queue, shifting the bottleneck entirely from the web container to the database pool.

3.6 ThreadLocal Semantics and Request Context

In enterprise Java, `java.lang.ThreadLocal` variables are ubiquitous for propagating request-scoped context (e.g., security tokens, transaction contexts, distributed trace IDs).

Under the 1:1 platform thread paradigm, the number of live platform threads was bounded by container configuration (e.g., 200 threads). Storing context in `ThreadLocal` incurred a predictable, bounded memory overhead:

$$M_{\text{context}} = N_{\text{threads}} \times S_{\text{entry}} \tag{3.2}$$

Under Virtual Threads, where an application can instantiate 100 000 concurrent threads simultaneously, retaining large objects in `ThreadLocal` variables without immediate, deterministic cleanup creates severe memory retention hazards:

$$M_{\text{context}} = 100\,000 \times S_{\text{entry}} \tag{3.3}$$

If $S_{\text{entry}} = 50$ KB, memory consumption scales to 5 GB, inducing garbage collection pressure and OutOfMemoryError exceptions. Furthermore, thread-local caching anti-patterns (such as pooling reusable buffers in `ThreadLocal`) fail catastrophically because virtual threads are ephemeral and never reused.

3.7 Foreign Function Boundary: JNI and Native Execution

When Java applications interface with native platform libraries (C/C++) via the Java Native Interface (JNI) or Project Panama Foreign Function & Memory (FFM) API (JEP 454 [9]), execution crosses from the managed JVM runtime into unmanaged OS native space.

3.7.1 Carrier Captivity in Native Downcalls

When a virtual thread executes a JNI native method, HotSpot transitions the thread's execution state from `_thread_in_Java` to `_thread_in_native`. In this state:

- Native C code executes directly on the OS platform thread acting as the virtual thread's carrier.
- If the native code executes a blocking system call (e.g., POSIX `nanosleep()`, synchronous file I/O, or blocking socket read), the OS kernel suspends the underlying platform thread.
- Because native code executes outside JVM bytecode interpretation, HotSpot continuation yield hooks are completely bypassed. The virtual thread **cannot unmount**.
- The carrier thread remains captured for the entire duration of the native blocking call, starving the virtual thread scheduler.

3.7.2 The Observability Vacuum

Because JNI carrier capture bypasses the HotSpot continuation freeze state machine, HotSpot's diagnostic instrumentation is never triggered. Specifically:

- HotSpot generates **zero** `jdk.VirtualThreadPinned` events.
- Thread dumps report the virtual thread in state `RUNNABLE` (in native).
- OS process telemetry reveals low CPU utilization if the thread is suspended in a kernel wait state (e.g., Darwin/Mach `semaphore_wait_trap`).

This combination creates a dangerous diagnostic blind spot in enterprise monitoring systems.

3.8 Observability, Diagnostics, and Telemetry

Accurately observing virtual thread behavior requires coordinating three distinct diagnostic tiers:

1. **Java Flight Recorder (JFR):** HotSpot provides continuous event tracing. The key event for detecting monitor pinning is `jdk.VirtualThreadPinned`. Crucially, this event is recorded only when the pinning duration exceeds a configurable threshold:

   ```
   -XX:FlightRecorderOptions=stackdepth=64
   jdk.VirtualThreadPinned#threshold=20ms
   ```

   Pinning events with hold durations shorter than the threshold (< 20 ms) are discarded by the runtime to minimize diagnostic overhead.

2. **JVM Thread Dumps:** Traditional thread dumps (`jstack`) omit virtual threads by default due to the high overhead of formatting hundreds of thousands of threads. Virtual thread dumps require specialized tooling (such as `jcmd <pid> Thread.dump_to_file -format=json`).

3. **Operating System Process Telemetry:** Low-level OS telemetry (e.g., `ps`, `top`) captures physical CPU utilization and thread state transitions in the OS kernel scheduler.

3.9 Performance Foundations and Causal Queueing Models

Evaluating concurrency performance requires grounding observations in operational queueing theory:

3.9.1 Little's Law and Closed-Loop Systems

Little's Law [29] establishes that under stationary equilibrium, the average number of requests in a queueing system (L) equals the arrival rate (λ) multiplied by average residence time (W):

$$L = \lambda W \tag{3.4}$$

In a **closed-loop** system (such as Grafana k6 with fixed Virtual Users N and constant client think time Z):

$$N = X(R + Z) \tag{3.5}$$

where:

- N is the constant number of concurrent users (closed-loop population).
- X is the achieved throughput (req/s).
- R is the client-observed response time / residence time (s).
- Z is the client think time between consecutive requests (s).

Rearranging Equation 3.5 yields the theoretical response time:

$$R = \frac{N}{X} - Z \tag{3.6}$$

When throughput X plateaus at a system bottleneck, increasing the user population N linearly inflates response time R.

3.9.2 Amdahl's Law and Single-Server Serialization

Amdahl's Law [1] defines the maximum speedup S attainable by parallelizing a workload with a strictly serial fraction s:

$$S(N) = \frac{1}{s + \frac{1-s}{N}} \tag{3.7}$$

As concurrency $N \to \infty$, the maximum speedup asymptotically approaches $1/s$.

When an application contains a critical section protected by mutual exclusion with service duration D:

$$\mu = \frac{1}{D} \tag{3.8}$$

The system's maximum service capacity is fundamentally constrained to μ req/s. In this regime, unmounting virtual threads or expanding carrier pools cannot exceed μ; mutual exclusion enforces strict sequential processing through the critical section.

Chapter 4

Research Design and Methodology

4.1 Research Design and Mode

This dissertation adopts an **empirical systems engineering** paradigm [42, 48] to evaluate the architectural, performance, and runtime consequences of migrating a Spring Boot Model-View-Controller (MVC) system to Java 21 Virtual Threads (Project Loom). Rather than treating runtime migration as a trivial configuration toggle, this research models the transition as an architectural intervention within a multi-tier concurrency and queueing network.

The empirical investigation is structured around a controlled, instrumented Spring Boot MVC benchmark application (`virtual-threads-demo`). In empirical software engineering, research designs must balance internal validity (the degree to which observed outcomes are directly attributable to experimental treatments) against external validity (the generalizability of conclusions to industrial systems):

- **Limits of Synthetic Micro-benchmarks:** Instruction-level benchmarks implemented via the Java Microbenchmark Harness (JMH) excel at isolating CPU cache and branch behavior, but completely omit framework-level interactions: HTTP socket multiplexing, servlet container request dispatching via `newVirtualThreadPerTaskExecutor()`, connection pool checkout arbitration, and request-scoped `ThreadLocal` lifecycles.

- **Limits of Production Deployment Testing:** Full-scale production systems introduce severe confounding environmental variables—wide-area network jitter, database optimizer variance, multi-tenant CPU throttling, and background oper-

ating system daemons—that obscure the low-level scheduling and synchronization mechanics of the JVM.

- **The Representative Testbed Strategy:** A dedicated, representative framework artifact provides an optimal balance: it preserves the authentic request lifecycle of Spring MVC, Apache Tomcat, and HikariCP while establishing strict experimental controls over carrier concurrency, workload generation, lock boundaries, and flight recorder telemetry.

4.2 Question-to-Method Operationalisation

To transition the canonical research questions (RQ1 and RQ2) into empirically verifiable hypotheses, each question is operationalized around explicit treatments, controlled baselines, dependent metrics, and decision rules. Table 4.1 details this operational mapping.

Table 4.1: Research-question-to-method operationalisation matrix.

RQ	Experimental Treatment	Controlled Baseline	Dependent Metrics	Operational Decision Rule
RQ1	Virtual Threads + `synchronized`	Platform Threads + `synchronized`	Throughput (X), $p50$, $p95$, $p99$ latency, JFR pins	Carrier pinning confirmed if JFR records events $\geq$ 20 ms; carrier starvation confirmed if throughput collapses under `parallelism=1`.
RQ1	Virtual Threads + `ReentrantLock`	Virtual Threads + `synchronized`	Throughput (X), latency distributions, σ dispersion	Unpinning confirmed if JFR events = 0; Amdahl serialization confirmed if throughput remains identical (1.956 vs. 1.967 req/s).
RQ1	AQS non-fair lock dispersion	Virtual Threads + `synchronized`	Latency standard deviation (σ) and range (Δ)	AQS direct-CAS barging confirmed if `ReentrantLock` dispersion exceeds `synchronized` by $> 5\times$ (σ = 16.53 ms vs. 0.70 ms).
RQ1	Java 24 JEP 491 runtime	Java 21 LTS Virtual Threads + `synchronized`	JFR pinning count, throughput (X)	JEP 491 monitor unpinning confirmed if JFR pinning events drop from 108 to 0 while throughput remains bounded at $\approx$ 1.97 req/s.
RQ2	Native JNI POSIX `nanosleep()`	Pure Java managed `Thread.sleep()`	Throughput (X), JFR pinning count, process %CPU	Carrier capture confirmed if throughput drops to 1.97 req/s; observability blind spot confirmed if JFR pinning is strictly 0 and CPU is 1.04% $\pm$ 0.12%.
RQ2	Persistence holding (Stratum 8)	Pre-acquisition delay (Treatment A) & Pool 10	Throughput (X), HikariCP active/pending queues	Pool saturation confirmed if Pool 1 caps throughput to 3.27 req/s with 49 queued VUs, while Pool 10 scales to 32.60 req/s with 0 JFR pins.

4.3 Materials, Data, Tools, and Provenance

The experimental testbed comprises open-source software libraries, standard compilation toolchains, and dedicated bare-metal hardware:

- **Hardware Platform:** Dedicated bare-metal Apple M1 processor (8 physical execution cores: 4 Firestorm performance cores, 4 Icestorm efficiency cores), 8 GB unified LPDDR4X memory, Darwin 25.2.0 (macOS 15.2, arm64).
- **Software Stack:** Java 21 LTS (Azul Zulu 21.0.8, OpenJDK HotSpot 64-bit Server VM), early-access OpenJDK 24.0.2 (build 24+36), Spring Boot 3.5.12, embedded Apache Tomcat 10.1.52, HikariCP 6.3.3, and PostgreSQL 42.7.10 JDBC driver.
- **Workload Generator:** Grafana k6 (v1.7.1) [17] compiled for Darwin arm64.
- **Telemetry Daemons:** JDK Flight Recorder (JFR) loaded via HotSpot `profile.jfc`, with a BSD `ps` polling daemon sampling process CPU utilization at 1.0-second intervals.

4.3.1 Repository Evidence Strata Census

To maintain forensic traceability and prevent conflating exploratory runs with ratified benchmarks, the complete corpus of 125 physical trials (+1 schema template) is partitioned into eight non-overlapping strata, as detailed in Table 4.2.

Table 4.2: Master repository execution census across eight evidence strata.

Stratum	Focus Description	Trials	Admitted Status	Epistemic Role
Stratum 1	Java 21 Primary Synchronization Matrix	12	Admitted (Primary)	Mutex treatments (`synchronized`, `ReentrantLock`, no-lock)
Stratum 2	Java 24/25 Runtime Evolution Matrix	10	Admitted (Supporting)	JEP 491 object monitor unpinning verification
Stratum 3	Multi-JDK Native JNI Boundary Matrix	27	Admitted (Primary)	$3 \times 3 \times 3$ factorial evaluation of JNI POSIX sleep
Stratum 4	Supporting Architectural Controls	18	Admitted (Supporting)	Endpoint controls: `/orders`, `/products`, `/cpu`
Stratum 5	8-Carrier Historical Baseline Matrix	12	Admitted (Baseline)	Unconstrained 8-carrier Apple M1 baseline
Stratum 6	Exploratory Pilots & Diagnostic Runs	27	Quarantined	Preliminary lock-/semaphore exploratory runs
Stratum 7	Construct-Flawed / Defective Runs	13	Quarantined	Pre-acquisition sleep flaws, unmanifested runs
Stratum 8	Two-Treatment Persistence Ablation	6	Admitted (Primary)	HikariCP pool size 1 vs. 10 connection-holding ablation
Template	Run Manifest Reference Schema	1	Metadata Only	Canonical JSON schema specification
Total	**Complete Repository Inventory**	**126**	**504 Artifacts**	**Cryptographically SHA-256 verified**

4.4 Workload Protocol and Closed-Loop Queueing Mechanics

Benchmark load is generated using a staged closed-loop concurrency profile executed over 55.0 seconds per trial:

1. **Stage 1 (Ramp-Up, $t \in [0\ \mathbf{s}, 15\ \mathbf{s}]$):** Concurrency scales linearly from 0 to N Virtual Users (VUs), warming JIT compilation and populating thread pools.

2. **Stage 2 (Steady-State Hold, $t \in [15\ \mathbf{s}, 45\ \mathbf{s}]$):** Load is maintained at constant concurrency $N = 10$ VUs (Strata 1–5) or $N = 50$ VUs (Stratum 8) for exactly 30.0 seconds.

3. **Stage 3 (Ramp-Down, $t \in [45\ \mathbf{s}, 55\ \mathbf{s}]$):** Concurrency ramps down linearly from N to 0.

4.4.1 Closed-Loop vs. Open-Loop Mechanics

In accordance with Schroeder et al. [43], selecting a closed-loop workload model enforces client-side self-throttling. Each virtual user issues an HTTP request, awaits the response, executes a constant think-time delay ($Z = 100$ ms $= 0.100$ s), and then submits the subsequent request.

By Little's Law for interactive closed-loop systems [29]:

$$N = X \times (R + Z) \iff R = \frac{N}{X} - Z \tag{4.1}$$

where N is the active virtual user population, X is throughput (req/s), R is residence/response time (s), and Z is think time (s).

For a serialized critical section with duration $D = 0.500$ s, the theoretical service capacity ceiling is:

$$\mu = \frac{1}{D} = \frac{1}{0.500\ \text{s}} = 2.000\ \text{req/s} \tag{4.2}$$

Under $N = 10$ VUs and $Z = 0.100$ s, the theoretical saturated cycle time is:

$$R + Z = \frac{N}{\mu} = \frac{10}{2.000\ \text{req/s}} = 5.000\ \text{s} \implies R = 4.900\ \text{s} \tag{4.3}$$

Any experimental observation deviating from this theoretical expectation indicates either additional service overhead or queueing departure from stationary equilibrium.

4.5 Dual-Window Steady-State Analysis Standard

A central methodological contribution of this dissertation is the formalization of the **Dual-Window Reporting Standard** (addressing `CLM-METH-02`):

- **Whole-Run Window ($t \in [0\ \mathbf{s}, 55\ \mathbf{s}]$):** Computes metrics across all completed requests over the entire test duration (55.0 s denominator).

- **Steady-State Hold Window ($t \in [15\ \mathbf{s}, 45\ \mathbf{s}]$):** Isolates requests whose completion timestamps fall strictly within the 30-second constant-load hold stage (30.0 s denominator).

During the 15-second ramp-up stage, the server queue is sparsely populated, yielding transient response times between 500 ms and 3 900 ms. In ramp-sensitive workloads (such as `platform-synchronized`), aggregating over the whole run depresses median latency to $\approx 3\,920$ ms, creating a **25.8% under-reporting distortion** relative to the true steady-state queueing equilibrium ($\approx 4\,930$ ms). Reporting both windows side-by-side guarantees complete transparency and eliminates warm-up bias [44].

4.6 Telemetry Pipeline and Correctness Invariants

The experimental infrastructure coordinates three independent telemetry layers:

1. **Client Stream (k6):** Raw JSON lines streams recording microsecond-accurate request start timestamps, durations, and HTTP response codes.

2. **JVM Runtime Stream (JFR):** Continuous flight recordings capturing `jdk.VirtualThreadPinned` events. Crucially, HotSpot filters pinning events shorter than 20 ms (`threshold=20ms`). Thus, zero recorded events confirms the absence of pinning events $\geq$ 20 ms, but does not preclude sub-20ms micro-pins (`CLM-METH-04`).

3. **Host Telemetry Stream (OS):** Periodic process sampling recording overall JVM CPU utilization ($\%cpu$). During JNI POSIX sleep, process CPU averages $1.04\% \pm 0.12\%$, proving that the carrier thread is suspended in a kernel wait state rather than busy-spinning (`CLM-RQ2-02`).

Every trial admitted into the primary corpus must satisfy seven strict correctness gates: HTTP 200 status on 100% of requests, sequential consistency in transactional endpoints, clean zero-exit process termination, driver carrier-safety verification (`pgjdbc >= 42.6.0`), cryptographic SHA-256 manifest matching, and membership in pre-registered evidence strata.

4.7 Validity Controls and Methodological Mitigations

Table 4.3 summarizes the primary threats to validity and the concrete mitigations implemented in this research.

Table 4.3: Methodological threats to validity and experimental mitigations.

Validity Category	**Threat / Risk**	**Applied Mitigation and Boundary**
Construct Validity	Pre-acquisition delay flaw in legacy `/orders`	Resolved via Two-Treatment Factorial Persistence Ablation (Stratum 8): Treatment A establishes unheld baseline; Treatment B holds connection across blocking sleep.
Measurement Validity	JFR 20 ms event threshold in `profile.jfc`	Claims of "zero pinning" explicitly qualified to events ≥ 20 ms; unmounting verified via AQS code inspection.
Internal Validity	Single-carrier scheduler constraint (`parallelism=1`)	Applied deliberately for mechanism isolation; unconstrained multi-carrier scaling captured in Stratum 5 (8 carriers).
Internal Validity	Client self-throttling in closed-loop load	Analyzed explicitly via Little's Law ($N = X(R+Z)$); prevents network packet loss and client timeout artifacts.
External Validity	Apple Silicon hardware architecture (arm64)	Hardware characteristics documented in detail; conclusions bounded to JVM-internal concurrency mechanics.
Statistical Validity	Small sample size ($N = 3$ process repetitions)	Repetitions reported strictly as descriptive engineering bounds (σ, Δ); inferential testing (p-values, ANOVA) excluded.
Reproducibility	Historical glob expansion inventory discrepancy	Audited master inventory of 504 physical disk files with 100% SHA-256 checksum agreement on recorded comparisons.

4.8 Academic Ethics and AI Disclosure

In strict compliance with ELTE Faculty of Informatics Dean's Directive 1/2026 [12], generative AI assistance (Anthropic Claude 3.5 Sonnet) was utilized solely for prose polishing, structural outlining, and python script scaffolding. Zero empirical benchmark measurements, latency figures, or flight recordings were generated by artificial intelligence. All empirical findings originate from physical hardware executions and were independently verified through cryptographic hashing and mathematical recalculation.

Chapter 5

Implementation and Research Infrastructure

5.1 System Architecture and Application Anatomy

To evaluate the runtime interactions between Java 21 Virtual Threads, monitor pinning, mutual exclusion serialization, relational persistence, and foreign function boundaries, an instrumented benchmark application (`virtual-threads-demo`) was engineered. The system models an enterprise Model-View-Controller (MVC) microservice architecture implemented in Java 21 LTS utilizing the Spring Boot 3.5.12 framework [45], deployed on embedded Apache Tomcat 10.1.52 [3], and compiled via Apache Maven 3.9.

The source tree is organized under root package `com.thesis.virtualthreadsdemo`, adhering to a strict separation of concerns across four primary architectural layers: HTTP controllers, service implementations, persistence repositories, and native shared libraries. Table 5.1 details the empirical endpoint catalog and their corresponding experimental roles.

Table 5.1: Empirical endpoint catalog and experimental roles.

Endpoint	Method	Service Component	Primary Concurrency Mechanism	Target RQ
`/payments`	`POST`	`PaymentService`	Configurable mutex (`synchronized`, `ReentrantLock`, `none`) + 500 ms critical section	RQ1
`/orders`	`GET`	`OrderService`	`ThreadLocal` context + 300 ms simulated delay + HikariCP query	RQ2
`/products`	`GET`	`ProductService`	Fast-path non-blocking `JdbcTemplate` query (no artificial delay)	RQ1, RQ2
`/native`	`GET`	`NativeBlocking Service`	JNI binding to C POSIX `nanosleep(500ms)` syscall	RQ2
`/native-short`	`GET`	`NativeBlocking Service`	JNI binding to C `nativeNoop()` (transit baseline)	RQ2
`/cpu`	`GET`	`CpuService`	5×10^6 iterations of `Math.fma` and `Math.sqrt`	RQ1, RQ2
`/hello`	`GET`	`HelloController`	Static string allocation (zero-I/O health probe)	Diag.

5.1.1 Framework Bill of Materials and Pinned Dependencies

To eliminate version ambiguity, all runtime dependencies are statically pinned in `pom.xml`:

- **Java Runtime:** OpenJDK 64-Bit Server VM (Azul Zulu 21.0.8 / OpenJDK 21.0.8, bytecode version 21). Comparative replication trials utilized OpenJDK 24.0.2 (build 24+36).
- **Framework Stack:** Spring Boot 3.5.12 pulling Spring Framework 6.2.x.
- **Embedded Servlet Container:** Apache Tomcat 10.1.52.

- **Connection Pool:** HikariCP 6.3.3.
- **Relational Database Driver:** PostgreSQL JDBC Driver 42.7.10, safely exceeding the `pgjdbc >= 42.6.0` threshold required to guarantee that internal driver monitors do not pin carrier threads.

5.2 Tomcat Request Dispatch and Virtual Thread Wiring

In Spring Boot 3, activating Virtual Threads requires only declaring:

```
spring.threads.virtual.enabled: true
```

This property triggers `TomcatWebServerFactoryCustomizer` to configure Tomcat's protocol handler with `Executors.newVirtualThreadPerTaskExecutor()`. Table 5.2 compares the architectural request lifecycle between Virtual Thread mode and the baseline Platform Thread mode.

Table 5.2: Tomcat request dispatch architecture: Virtual vs. Platform concurrency.

Architectural Phase	**Virtual Threads** (`enabled: true`)	**Platform Threads** (`enabled: false`)
Connection Ingestion	Dedicated OS platform threads (`NioEndpoint` Acceptor/Poller via `kqueue`/`epoll`)	Dedicated OS platform threads (`NioEndpoint` Acceptor/Poller via `kqueue`/`epoll`)
Servlet Task Execution	Ephemeral `VirtualThread` per request (heap continuation $<$ 1 KB)	Bounded worker thread checked out from Tomcat pool (`maxThreads = 200`)
Scheduler Binding	HotSpot carrier pool (`ForkJoinPool-1-worker-N`)	Direct OS kernel thread scheduling
Blocking Behavior	Continuation unmounts from carrier thread on blocking I/O or AQS park	Carrier OS thread remains physically blocked in kernel space

5.3 Payment Synchronization Implementations

The primary synchronization treatments evaluated in RQ1 are encapsulated within `PaymentService`. The service models a financial transaction where state mutations must be serialized:

- `synchronized` **Treatment:** Synchronizes on a private `Object monitor` before invoking `simulatePaymentCall()` (500 ms simulated payment gateway sleep). In Java 21, this triggers HotSpot monitor pinning.
- `reentrant-lock` **Treatment:** Acquires an explicit `java.util.concurrent.locks.ReentrantLock`. Uncontended acquisition succeeds via atomic CAS; contended threads park via AQS and unmount cleanly.
- `fair-reentrant-lock` **Treatment:** Uses `new ReentrantLock(true)`, enforcing FIFO queue ordering and suppressing direct-CAS barging.
- `none` **Diagnostic Control:** Directly invokes `simulatePaymentCall()` without synchronization, establishing the theoretical upper throughput bound for concurrent execution under the simulated delay.

5.4 Foreign Function Interface: Native POSIX Shared Library

To evaluate unmanaged native execution (RQ2), a custom dynamic shared library (`libnativeblocking.dylib` on macOS; `.so` on Linux) was implemented in C (`native/native_blocking.c`) and bound via JNI to `NativeBlockingService`:

```
JNIEXPORT void JNICALL
Java_com_thesis_virtualthreadsdemo_service_NativeBlockingService_nativeSleep(
        JNIEnv *env, jobject self, jlong sleep_millis) {
    struct timespec remaining = {
        .tv_sec = (time_t) (sleep_millis / 1000),
        .tv_nsec = (long) ((sleep_millis % 1000) * 1000000L)
    };
    while (nanosleep(&remaining, &remaining) == -1 && errno == EINTR) {
        /* Resume bounded sleep if interrupted by a signal. */
    }
}
```

5.4.1 HotSpot Carrier Captivity and Diagnostic Omission

When `nativeSleep(500)` is invoked:

1. HotSpot transitions the carrier thread state from `_thread_in_Java` to `_thread_in_native`.

2. The POSIX `nanosleep()` system call places the physical OS platform thread into an interruptible Darwin/Mach wait state. The carrier thread yields its CPU core to the OS kernel scheduler.

3. Because execution occurs outside JVM bytecode interpretation, no bytecode safepoints are reached and `Continuation.yield()` is never called. The virtual thread remains rigidly bound to the carrier.

4. The JFR `jdk.VirtualThreadPinned` event triggers only when yield fails. Because yield is never attempted, JFR records strictly **0 pinning events** across all 27 native trials, creating a complete diagnostic blind spot.

5. Process telemetry sampled at 1.0-second intervals confirms that process CPU utilization remains at $1.04\% \pm 0.12\%$, disproving busy-wait myths.

5.5 Persistence Layer and HikariCP Connection Pooling

Relational persistence is implemented using Spring `JdbcTemplate` backed by HikariCP across an embedded H2 database and containerized PostgreSQL 15. The schema (`schema.sql`) defines two tables: `orders` and `products`, pre-populated at boot time.

5.5.1 Two-Treatment Factorial Persistence Design

To resolve the construct flaw identified in early exploratory trials (where simulated delay was uncoupled from connection leasing), `OrderService` was instrumented with a configurable persistence treatment flag (`demo.orders.hold-connection-during-delay`):

- **Treatment A (Pre-Acquisition Delay, Unheld Baseline):** The 300 ms delay executes *before* calling `jdbcTemplate.queryForList()`. The virtual thread holds zero database connections during sleep, unmounting cleanly. HikariCP pool occupancy is negligible ($L_{\text{pool}} \approx 0.0125$ connections), leaving the pool over 98% idle.

- **Treatment B (Connection-Held Delay, Saturated Contention):** The connection is acquired first via `jdbcTemplate.execute()`, and the 300 ms delay is held inside the active connection callback:

```
jdbcTemplate.execute((ConnectionCallback<...>) connection -> {
    simulateSlowQuery(); // Connection held for 300 ms!
    try (PreparedStatement ps = connection.prepareStatement(...)) { ... }
});
```

Under Treatment B, connection checkout duration is strictly $T_{\text{hold}} = 0.300$ s. By Little's Law, a pool size of $K = 1$ enforces a single-server bottleneck ($X_{\max} = 1/0.300 \approx 3.33$ req/s), causing 49 virtual users to queue in HikariCP. Expanding pool size to $K = 10$ scales throughput tenfold ($X_{\max} \approx 33.3$ req/s) with 0 JFR pinning events, validating the Downstream Bottleneck Shift Law.

5.6 Request Context and ThreadLocal Lifecycle

In `OrderService`, request-scoped context is stored in a static `ThreadLocal<String>` variable. In platform thread containers, memory overhead is bounded by pool size ($200 \times M_{\text{context}}$). Under Virtual Threads, where concurrency can reach 100 000 simultaneous threads, heap memory scales linearly:

$$\text{Memory}_{\text{virtual}} = N_{\text{concurrent}} \times M_{\text{context}} \tag{5.1}$$

If $M_{\text{context}} = 50$ KB (e.g., tokens, tracing spans), heap overhead reaches 5.0 GB. While `OrderService` enforces cleanup via `REQUEST_CONTEXT.remove()` in a `finally` block, thread-local caching anti-patterns represent a major architectural vulnerability under virtual threads.

5.7 Benchmarking Harness and Telemetry Pipeline

Benchmark execution was fully automated by `run-benchmark.sh`:

1. **JVM Initialization:** Launches the Spring Boot jar with JVM scheduler controls (`-Djdk.virtualThreadScheduler.parallelism=1`), JFR profiling options, and native library bindings.

2. **Health Verification:** Polls `/products` until receiving HTTP 200 OK before initiating load.
3. **Telemetry Daemon:** Launches a background subshell sampling `ps -p $APP_PID -o %cpu=` every 1.0 second.
4. **k6 Workload Execution:** Dispatches `k6/isolated-load-test.js` across 15s ramp-up, 30s steady hold, and 10s ramp-down.
5. **Manifest and Checksum Generation:** Captures process exit codes, records Git commit hashes (`1ccb617`, `5f9936c`), and computes SHA-256 digests of k6 JSON streams, JFR binary recordings, and telemetry CSV files into JSON run manifests.

5.8 Verification Suite and Artifact Provenance Tooling

The testbed is hardened with an automated JUnit 5 test suite executing 8 test suites with 100% passing assertions (`rtk mvn test`):

- `PaymentControllerTest`: Verifies mutual exclusion and sequential consistency across all lock modes.
- `OrderControllerTest`: Validates JDBC queries and `ThreadLocal` context cleanup.
- `NativeControllerTest`: Validates JNI shared library linking and syscall return codes.
- `ProductControllerTest` & `CpuControllerTest`: Verify fast-path persistence and compute workloads.

Furthermore, three standalone Python tools ensure complete reproducibility:

- `scripts/generate_artifact_inventory.py`: Audits all 504 disk files in `results/`, generating the master ledger `docs/evidence-freeze/artifact-inventory.csv`.
- `scripts/generate_run_family_matrix.py`: Parses manifests to classify all 126 records into the 8 strata.
- `scripts/recalculate_metrics.py`: Streams through 150,000 raw k6 JSON lines to compute whole-run vs. steady-state hold-window metrics.
- `scripts/validate_normalized_evidence.py`: Verifies the consolidated JSON evidence schema against cryptographic SHA-256 digests.

Chapter 6

Experimental Results

6.1 Evidence Inventory and Provenance Ledger

In accordance with empirical systems software engineering standards, this chapter documents directly measured observations, event logs, operating system telemetry streams, and descriptive distributions without speculative interpretation. All performance metrics adhere to the **Dual-Window Reporting Mandate** (`CLM-METH-02`), reporting side-by-side:

1. **Whole-Run Metrics ($t \in [0\ \mathbf{s}, 55\ \mathbf{s}]$):** Full execution window, encompassing client ramp-up and ramp-down (55.0 s denominator).

2. **Steady-State Hold-Window Metrics ($t \in [15\ \mathbf{s}, 45\ \mathbf{s}]$):** Filtered execution window, isolating the 30-second constant-concurrency hold stage (30.0 s denominator).

The experimental evidence analyzed comprises **504 physical disk files** totaling 965.99 **MB ($\approx$ 0.94 GB)** under `results/`. This population represents **125 physical benchmark executions** across eight non-overlapping strata and one schema reference template. Automated cryptographic SHA-256 validation confirmed 100% checksum agreement across all recorded comparisons.

Table 6.1 establishes the formal Result Provenance Ledger, tracing each central claim to its experimental method, raw evidence files, and frozen repository location.

Table 6.1: Master result provenance ledger mapping claims to empirical artifacts.

Claim / Result	**Method and Input**	**Raw Evidence and Metric**	**Corpus Location**
Carrier Pinning Invariant (`CLM-RQ1-01`)	Closed-loop 10 VUs, `synchronized`, Java 21	JFR `jdk.VirtualThreadPinned` events $\geq$ 20 ms; exactly 108 events	Stratum 1, 3 trials (`20261015-142222` to `142430`)
Mutex Unpinning (`CLM-RQ1-03`)	Closed-loop 10 VUs, `ReentrantLock`, Java 21	JFR event stream; exactly 0 pinning events recorded	Stratum 1, 3 trials (`20261015-142534` to `142742`)
Amdahl Throughput Invariance (`CLM-RQ1-04`)	Factorial comparison: `synchronized` vs. `ReentrantLock`	Steady-state throughput: 1.967 vs. 1.956 req/s ($< 0.6\%$ delta)	Stratum 1, 6 trials; Table 6.2
AQS Barging Dispersion (`CLM-RQ1-05`)	$N = 3$ process repetitions; trial-level variance	Median latency standard deviation: $\sigma = 16.53$ ms vs. 0.70 ms ($> 23\times$)	Stratum 1 & Stratum 5; Table 6.3
JEP 491 Monitor Unpinning (`CLM-RQ1-06`)	Java 24.0.2 vs. Java 21 LTS `synchronized`	JFR pinning events drop from 108 to 0; throughput caps at 1.967 req/s	Stratum 2, 6 trials; Table 6.4
JNI Carrier Captivity (`CLM-RQ2-01`, `CLM-RQ2-02`)	Native C POSIX `nanosleep(500ms)` downcall	Steady throughput 1.967 req/s, 0 JFR pins, process CPU $1.04\% \pm 0.12\%$	Stratum 3, 9 trials; Table 6.5
Persistence Bottleneck Shift (`CLM-RQ2-03`)	Two-Treatment ablation: HikariCP pool 1 vs. 10	Pool 1: 3.27 req/s, 15.1 s $p50$, 49 queued VUs; Pool 10: 32.60 req/s, 1.43 s $p50$, 0 pins	Stratum 8, 6 trials; Table 6.6

6.1.1 Semantic Invariant and Correctness Verification

Before analyzing performance distributions, all admitted trials were verified against the semantic correctness criteria: 100.0% of HTTP requests completed with status `200 OK` (324 requests in `payments-synchronized`, 324 in `payments-reentrant-lock`, 330 in `platform-synchronized`, 978 in `blocking-jni`, and 37,752 in `native-short`). The HTTP error rate was strictly **0.000%** across all primary trials. All application processes terminated cleanly with an exit code of 0 upon receiving termination signals.

6.2 Results for RQ1: Synchronization, Pinning, and Serialization

Table 6.2 reports the exact trial-level and aggregate metrics for Stratum 1 on OpenJDK 21.0.8 under single-carrier isolation (`parallelism=1`, `maxPoolSize=1`) with 10 closed-loop Virtual Users.

Table 6.2: Java 21 primary synchronization matrix performance (Stratum 1).

Run ID	Condition	Whole [req/s]	Steady [req/s]	$p50$ [ms]	$p95$ [ms]	Pins
`142222` (T1)	`synchronized`	1.964	1.967	4 967.12	4 981.29	**108**
`142326` (T2)	`synchronized`	1.964	1.967	4 967.26	4 989.23	**108**
`142430` (T3)	`synchronized`	1.964	1.967	4 965.98	4 979.47	**108**
`synchronized` **Mean**	—	**1.964**	**1.967**	**4 966.79**	**4 983.33**	**108.0**
`142534` (T1)	`reentrant-lock`	1.964	1.967	4 962.92	4 974.96	**0**
`142638` (T2)	`reentrant-lock`	1.964	1.933	4 992.70	5 003.07	**0**
`142742` (T3)	`reentrant-lock`	1.964	1.967	4 990.23	5 015.33	**0**
`reentrant-lock` **Mean**	—	**1.964**	**1.956**	**4 981.95**	**4 997.78**	**0.0**
`143422` (T1)	`platform-sync`	2.000	1.967	4 928.54	8 972.52	**0** (N/A)
`143526` (T2)	`platform-sync`	2.000	1.967	4 922.60	8 975.80	**0** (N/A)
`143631` (T3)	`platform-sync`	2.000	1.967	4 937.33	8 980.64	**0** (N/A)
`platform-sync` **Mean**	—	**2.000**	**1.967**	**4 929.49**	**8 976.32**	**0.0**
`no-lock` **Mean**	Diagnostic Control	**12.964**	**16.489**	**506.61**	**509.32**	**0.0**

6.2.1 Key RQ1 Empirical Invariants

1. **Monitor Pinning Invariant (`CLM-RQ1-01`):** In every synchronized trial, JFR recorded exactly **108** qualifying `jdk.VirtualThreadPinned` events, matching completed HTTP transactions 1:1.

2. **Suppression of Pinning via `ReentrantLock`** (`CLM-RQ1-03`): Replacing `synchronized` with `ReentrantLock` reduced qualifying pinning events from 108 to strictly **0**.

3. **Throughput Invariance Across Mutex Implementations (`CLM-RQ1-04`):** Steady-state throughput remained identical at 1.967 **req/s** for `synchronized` versus 1.956 **req/s** for `reentrant-lock` (a minor delta of 0.011 req/s or 0.56%). Both match the single-server Amdahl ceiling ($\mu = 1/0.500$ s $= 2.0$ req/s).

4. **The Ramp-Up Distortion (`CLM-METH-02`):** In `platform-synchronized`, whole-run median latency was 3 917.56 ms, whereas steady-state hold median latency was 4 929.49 ms. Incorporating the uncrowded 15-second ramp-up artificially depressed median latency by 1 011.93 **ms** (−25.83%).

5. **Diagnostic Upper Bound (`no-lock`):** Eliminating mutual exclusion increased throughput from 1.967 req/s to 16.489 **req/s** and reduced median latency to 506.61 **ms**.

6.2.2 Trial-Level Dispersion and the AQS Barging Signature (`CLM-RQ1-05`)

Table 6.3 details the trial-level dispersion metrics calculated across the $N = 3$ process repetitions.

Table 6.3: Steady-state trial-level dispersion across $N = 3$ process repetitions.

Condition	Metric	T1 [ms]	T2 [ms]	T3 [ms]	Range Δ [ms]	Std Dev σ [ms]
`synchronized` (S1)	Steady $p50$	4 967.12	4 967.26	4 965.98	**1.28**	**0.70**
`synchronized` (S1)	Steady $p95$	4 981.29	4 989.23	4 979.47	9.77	5.19
`reentrant-lock` (S1)	Steady $p50$	4 962.92	4 992.70	4 990.23	**29.78**	**16.53**
`reentrant-lock` (S1)	Steady $p95$	4 974.96	5 003.07	5 015.33	40.37	20.70
`platform-sync` (S1)	Steady $p50$	4 928.54	4 922.60	4 937.33	14.73	7.41
`no-lock` (Diagnostic, S1)	Steady $p50$	506.63	506.69	506.50	0.20	0.10
`synchronized` (S5, 8-carrier)	Steady $p50$	4 927.80	4 930.96	4 935.03	7.24	3.63
`reentrant-lock` (S5, 8-carrier)	Steady $p50$	4 936.77	4 938.60	4 968.95	32.18	18.07

Under single-carrier scheduling, `synchronized` exhibited an exceptionally narrow median latency range of $\Delta = 1.28$ ms ($\sigma = 0.70$ ms), whereas `reentrant-lock` exhibited $\Delta = 29.78$ ms ($\sigma = 16.53$ ms). This constitutes a **23-fold variance difference** ($\sigma = 16.53$ ms vs. 0.70 ms), consistent with the direct-CAS barging behavior of AQS `NonfairSync`.

6.2.3 Java 24 Runtime Evolution: JEP 491 Unpinning (Stratum 2)

Table 6.4 compares the runtime performance of Java 21 LTS against early-access OpenJDK 24.0.2 under identical mutual exclusion workloads.

Table 6.4: Java 21 vs. Java 24 monitor synchronization performance (Stratum 2).

JDK Runtime	Condition	Whole [req/s]	Steady [req/s]	$p50$ [ms]	$p95$ [ms]	Pins
OpenJDK 21.0.8	`synchronized`	1.964	1.967	4 966.79	4 983.33	**108** (Pinned)
OpenJDK 21.0.8	`reentrant-lock`	1.964	1.956	4 981.95	4 997.78	**0** (Unmounted)
OpenJDK 24.0.2	`synchronized`	2.000	1.967	4 944.67	**9 000.77**	**0** (JEP 491)
OpenJDK 24.0.2	`reentrant-lock`	1.982	1.967	4 952.60	4 968.27	**0** (Unmounted)

On Java 24, JEP 491 completely eliminated monitor pinning events (**108 to 0**), yet steady-state throughput remained strictly constrained at 1.967 **req/s**, confirming that unpinning does not bypass Amdahl's serialization law.

6.3 Results for RQ2: Architectural Migration Risks

6.3.1 Foreign Function Interface: JNI Native Boundary Results (Stratum 3)

Table 6.5 summarizes the results across nine repeated trials of `blocking-jni` (500 ms POSIX `nanosleep()`) on OpenJDK 21.0.8 under single-carrier scheduling.

Table 6.5: JNI foreign function execution telemetry (Stratum 3, $N = 9$).

Execution Condition	Steady [req/s]	$p50$ [ms]	Pins	Avg CPU	Carrier OS State
`blocking-jni` (9-run Mean)	**1.967**	**4 963.36**	**0**	**1.04% ± 0.12%**	Darwin/Mach Kernel Wait
`native-short` (Baseline)	1 143.9	0.78	0	38.2% ± 4.1%	Active User Execution
`pure-java-sleep` (Control)	1.967	4 956.1	0	1.1% ± 0.1%	HotSpot Yield / Parked

The empirical findings establish that blocking POSIX system calls silently capture the carrier thread: throughput collapses to 1.967 **req/s** with **exactly 0 JFR pinning events**, while process CPU consumption remains negligible at 1.04% ± 0.12% (peak steady-state $\leq 8.5\%$).

6.3.2 Two-Treatment Factorial Persistence Ablation (Stratum 8)

Table 6.6 details the results of the Two-Treatment Factorial Persistence Ablation under 50 concurrent virtual users across HikariCP pool sizes 1 and 10.

Table 6.6: Two-Treatment Factorial Persistence Ablation performance (Stratum 8).

Treatment	Pool	Whole [req/s]	Steady [req/s]	$p50$ [ms]	Act.	Pend.	Pins
Treatment A (Pre-Acquisition)	$K = 1$	96.26	123.10	304.04	0.01	0	0
Treatment A (Pre-Acquisition)	$K = 10$	96.17	123.02	304.63	0.01	0	0
Treatment B (Connection-Held)	$K = 1$	3.87	**3.27**	**15 123.0**	1.0	**49**	**0**
Treatment B (Connection-Held)	$K = 10$	30.85	**32.60**	**1 431.3**	9.2	**40**	**0**

Under Treatment B with Pool Size 1, throughput was strictly bounded across all trials to 3.27 **req/s** ($N = 3$, $\sigma = 0.00$ req/s), mathematically converging with Little's Law ($\mu = 1/0.300$ s $= 3.333$ req/s within 1.9%). Expanding pool size to 10 scaled throughput tenfold to 32.60 **req/s** (10.5× latency reduction). In all trials, JFR recorded **0 pinning events**, validating that virtual threads unmount cleanly while queue formation shifts entirely into the connection pool.

6.3.3 Supporting Architectural Controls (Stratum 4)

Table 6.7 reports baseline metrics for Stratum 4 diagnostic controls.

Table 6.7: Supporting architectural diagnostic controls (Stratum 4).

Endpoint	Thread Mode	Total Reqs	Steady [req/s]	$p50$ [ms]	$p95$ [ms]	Pins
`/products`	Virtual	12 502	96.98	2.33	6.57	0
`/products`	Platform	12 420	96.29	2.61	5.88	0
`/cpu`	Virtual	11 671	90.54	9.03	13.21	0
`/cpu`	Platform	11 188	86.61	13.79	20.24	0
`/orders` (A)	Virtual	3 144	24.11	311.35	318.56	0
`/orders` (A)	Platform	3 172	24.44	307.11	311.56	0

6.4 Historical Baselines and Flawed Strata (Strata 5 and 7)

In Stratum 5 (historical 8-carrier baseline on Apple M1), unconstrained execution confirmed that `synchronized` generated 110 pinning events and achieved 1.967 req/s, while `reentrant-lock` generated 0 pinning events and achieved 1.967 req/s. In Stratum 7 (containerized Docker PostgreSQL, 31,274 requests evaluated), executions omitted JFR flags; while HTTP latencies were stable ($p50 \approx 303.6$ ms), all claims regarding carrier unmounting on containerized PostgreSQL are declared Not Established (`CLM-RQ2-05`).

6.5 Negative, Null, and Ablation Findings

Table 6.8 systematically records all investigated mechanisms where expected performance gains failed to materialize.

Table 6.8: Systematic negative, null, and ablation findings ledger.

Investigated Mechanism	**Hypothesized Effect**	**Actual Empirical Observation**
`ReentrantLock` Refactoring	Eliminating pinning increases throughput	Throughput identical at 1.96 req/s ($< 0.5\%$ delta) due to Amdahl domain serialization.
Java 24 JEP 491 Unpinning	JDK 24 unpinning increases throughput	Throughput identical at 1.97 req/s; serialization persisted.
Foreign JNI Pinning Alerts	JNI blocking emits JFR pinning events	Exactly 0 pinning events emitted across 27 trials; complete diagnostic omission.
JNI Carrier CPU Consumption	Carrier blocked in native code burns 100% CPU	Process CPU remained at $1.04\% \pm 0.12\%$; thread suspended in Darwin/Mach wait state.
Database Pool Sensitivity	Constraining pool size always causes queueing	In Treatment A, pool was $>$ 98% idle (304 ms latency); queueing occurs only when connections are held across blocking delays (Treatment B).
Dynamic Carrier Compensation	HotSpot spawns extra carriers for pinned monitors	ForkJoinPool never spawned extra carriers; scheduler starved under single-carrier constraint.

Chapter 7

Analysis, Discussion, and Migration Guidance

7.1 Explicit Answers to the Research Questions

Synthesizing the empirical observations across all 125 physical trials and theoretical derivations, this section formally closes the canonical research questions. Table 7.1 details the Research Question Closure Ledger.

Table 7.1: Research-question closure ledger.

RQ	Evidence-Backed Answer	Strength & Uncertainty	Limit or Condition	Link
RQ1	In Java 21, `synchronized` blocks enclosing blocking I/O pin virtual threads to carriers (108 JFR events), starving the scheduler under single-carrier constraint. Replacing `synchronized` with `ReentrantLock` completely eliminates pinning (0 JFR events), but yields **identical throughput** (1.956 vs. 1.967 req/s) because Amdahl's Law enforces a service ceiling ($\mu = 1/D = 2.0$ req/s). Non-fair `ReentrantLock` direct-CAS barging inflates tail-latency standard deviation by a factor of 23 ($\sigma = 16.53$ ms vs. 0.70 ms). Java 24 (JEP 491) unpins monitors (0 events) while throughput remains bounded.	High internal validity; verified across 12 primary trials and JDK 24 boundary trials. Dispersion based on $N = 3$ descriptive process repetitions.	Bounded to single-carrier scheduler isolation; does not evaluate open-loop Poisson traffic.	C1
RQ2	Four primary architectural risks must be assessed before enabling Virtual Threads: (1) **Foreign JNI Syscalls:** Blocking POSIX `nanosleep()` holds carriers captive in Darwin/Mach wait states (1.04% ± 0.12% CPU) with zero JFR pinning alerts. (2) **Persistence Queue Shift:** Holding database connections across blocking operations unmounts virtual threads (0 pins), but throughput is capped by pool size ($X_{\max} = K/T_{\text{hold}}$), shifting queues entirely into HikariCP. (3) **Context Scaling:** Millions of virtual threads retaining state in `ThreadLocal` create $O(N)$ heap bloat. (4) **Diagnostic Blind Spots:** JFR filters pins < 20 ms, while `ThreadMXBean` omits virtual threads entirely.	Empirical discovery verified across 27 JNI trials, 6 Stratum 8 trials, and low-level HotSpot C++ source audit.	Specific to evaluated Spring Boot and HikariCP configurations; heap growth is an analytical projection.	C2, C3, C4

7.2 Theoretical Foundations: The Three-Tier Concurrency Taxonomy

To provide a sound conceptual framework for virtualized runtime concurrency, this dissertation establishes the **Three-Tier Concurrency Taxonomy**:

1. **Tier 1: Runtime Schedulability (Project Loom / Carrier Multiplexing):**
   - *Governing Principle:* $M : N$ user-mode thread multiplexing managed by OpenJDK HotSpot's `ForkJoinPool`.
   - *Architectural Problem Solved:* Eliminates OS-level platform thread memory overhead (1 024 KB stack reservation) and web container thread-pool exhaustion.
   - *Runtime Vulnerabilities:* Monitor pinning in Java 21 LTS (JEP 444 [40]) and silent foreign-function carrier captivity (JNI).
2. **Tier 2: Domain Synchronization (Software Architecture / Mutual Exclusion):**
   - *Governing Principle:* Amdahl's Law [1] and Bernstein's Conditions.
   - *Architectural Problem Solved:* Guarantees shared-state transactional integrity and sequential consistency.
   - *Fundamental Invariant:* Unpinning carrier threads **cannot unserialize** an application critical section. If service time is dominated by a serial duration D, maximum system service rate is immutable: $\mu = 1/D$.
3. **Tier 3: Backend Capacity (Infrastructure / Finite Resources):**
   - *Governing Principle:* Little's Law [29] and the Universal Scalability Law.
   - *Architectural Problem Solved:* Preserves downstream physical hardware capacity (relational database CPU, I/O, locks, and connection pools).
   - *Architectural Risk:* Multiplying application-tier thread concurrency without downstream admission control shifts queue accumulation into connection pool checkout queues ($X_{\max} = K/T_{\text{hold}}$).

7.3 The Authoritative Engineering Migration Framework

Translating these findings into industrial engineering practice yields an authoritative migration framework comprising five mandatory architectural rules, four correctness-preserving refactoring patterns, and a deterministic migration decision tree.

7.3.1 The Five Core Architectural Migration Rules

1. **Rule 1: The Mutex Refactoring Rule (`CLM-MIG-01`).** Enclosing blocking operations within `synchronized` blocks must be refactored to `ReentrantLock` in Java 21 to prevent carrier pinning. However, engineering teams must not expect throughput improvements if the critical section is serialized; unpinning restores carrier availability for unrelated requests, but cannot accelerate contended execution paths.

2. **Rule 2: The Native JNI Isolation Rule (`CLM-MIG-02`).** Blocking foreign function downcalls (JNI / POSIX syscalls) must never execute directly on virtual threads. They must be offloaded to dedicated, bounded platform thread pools.

3. **Rule 3: The Connection Pool Sizing Fallacy (`CLM-MIG-03`).** Database connection pools must **never** be inflated to match virtual thread concurrency. Connection pools must be sized strictly to database engine core capacity. Concurrency bounding must be implemented via application-level semaphores, and connection holding time (T_{hold}) must be minimized.

4. **Rule 4: The Production Observability Rule (`CLM-MIG-04`).** Production telemetry must not rely on legacy `ThreadMXBean` metrics. Teams must deploy continuous JFR profiling with tuned thresholds (`threshold=10ms` or `0ms`), monitor HikariCP active/pending connections, and track host OS thread states.

5. **Rule 5: `ThreadLocal` Memory Hygiene (`CLM-RQ2-07`).** Static `ThreadLocal` variables must enforce deterministic cleanup inside `finally` blocks immediately after request processing to prevent $O(N)$ heap inflation. Applications targeting Java 25 should migrate to Scoped Values (JEP 506 [18]).

7.3.2 Correctness-Preserving Refactoring Patterns

Pattern 1: Safe Mutex Refactoring (Eliminating Monitor Pinning). To resolve Java 21 monitor pinning without altering synchronization semantics:

```
// BEFORE: Pinned in Java 21
public synchronized void executeTransaction() {
    blockingCall();
}

// AFTER: Unpins cleanly via AQS
private final ReentrantLock lock = new ReentrantLock();

public void executeTransaction() {
    lock.lock();
    try {
        blockingCall();
    } finally {
        lock.unlock();
    }
}
```

Pattern 2: JNI Platform Thread Offloading. To isolate unmanaged native syscalls from the virtual thread carrier pool:

```
@Service
public class NativeIsolationService {
    private final ExecutorService nativeExecutor =
        Executors.newFixedThreadPool(
            Runtime.getRuntime().availableProcessors());

    public CompletableFuture<String> executeNativeCall() {
        return CompletableFuture.supplyAsync(() -> {
            return nativeBlockingService.block();
        }, nativeExecutor);
    }
```

```
}
```

Pattern 3: Minimizing Relational Connection Holding. To prevent shifting queues into HikariCP, connection checkout must be delayed until strictly necessary:

```
// ANTI-PATTERN: Holding connection across independent blocking I/O
@Transactional
public void processOrder() {
    externalHttpValidation(); // Connection held for 300 ms!
    orderRepository.save(order);
}

// CORRECT: Isolate connection lease strictly to database execution
public void processOrder() {
    externalHttpValidation(); // Zero database connections leased
    transactionTemplate.execute(status -> {
        return orderRepository.save(order); // Connection leased < 1 ms
    });
}
```

7.3.3 Master Migration Decision Matrix and Decision Tree

Table 7.2 summarizes the architectural triage rules for determining virtual thread eligibility.

Table 7.2: Architectural migration triage and action matrix.

Workload / Boundary	**Loom Compatibility**	**Mandatory Engineering Action**
Non-blocking / Stateless I/O	Fully Compatible	Enable `spring.threads.virtual.enabled: true`; immediate scalability benefit.
Synchronized Mutex Blocks	Hazardous in Java 21	Refactor to `ReentrantLock` (Pattern 1); evaluate tail-latency SLAs for AQS barging.
JNI / Foreign C Syscalls	Critical Starvation Risk	Offload syscalls to dedicated platform thread pool (Pattern 2).
Relational JDBC Persistence	Compatible with Isolation	Verify `pgjdbc >= 42.6.0`; do not inflate connection pool; minimize hold time (Pattern 3).
ThreadLocal Context Caching	Severe Memory Risk	Enforce `remove()` in `finally`; prepare migration to Scoped Values.

7.4 Master Threats to Validity

Table 7.3 provides an exhaustive threats-to-validity evaluation, categorized across four classical empirical dimensions [48].

Table 7.3: Master threats to validity matrix.

Dimension	Identified Threat	Methodological Mitigation and Scope
Internal Validity	Single-carrier scheduler constraint (`parallelism=1`)	Applied deliberately for mechanism isolation to unmask pinning; unconstrained 8-carrier scaling verified separately in Stratum 5.
Internal Validity	Closed-loop k6 workload dynamics	Formally derived via Little's Law ($R = N/X - Z$); findings represent stationary equilibrium rather than open-loop arrival bursts.
Internal Validity	JFR 20 ms event duration threshold	Claims of "zero pinning" qualified explicitly to events ≥ 20 ms; unmounting verified via AQS code inspection.
External Validity	Single bare-metal Apple Silicon host	Frame results as empirical systems study on arm64/Darwin; platform-independent JVM invariants separated from host timings.
External Validity	Homogeneous synthetic delays	Workloads evaluate physical mechanisms in isolation; migration guidance mandates staging under production traffic.
Construct Validity	Legacy `/orders` pre-acquisition delay flaw	Resolved via Two-Treatment Factorial Persistence Ablation in Stratum 8, contrasting unheld Treatment A against active connection-held Treatment B.
Construct Validity	Containerized PostgreSQL JFR omission	Formally declared **Not Established**; PostgreSQL carrier unmounting retained as external literature, not repository-measured.
Statistical Validity	Sample size ($N = 3$ process repetitions)	Repetitions reported strictly as descriptive engineering bounds (σ, Δ); inferential testing (ANOVA, p-values) excluded.

7.4.1 Transferability Boundary Audit

To assist practitioners, the transferability boundary matrix delineates findings that transfer directly to enterprise systems versus those requiring local calibration:

- **Directly Transferable Invariants:** Java 21 monitor pinning mechanics; decoupling carrier unpinning from Amdahl serialization; JNI silent carrier captivity; JFR threshold filtering; AQS barging dispersion.
- **Requires System-Specific Calibration:** Absolute throughput numbers (req/s) and latency percentiles; optimal HikariCP pool sizing; `ThreadLocal` heap pressure

under target request volume; multi-core scheduling under Linux CFS bandwidth quotas.

7.5 Engineering, Operational, and Academic Relevance

7.5.1 Architectural and Operational Relevance

Enabling virtual threads removes the implicit admission control historically provided by bounded platform thread pools. Software architects must introduce explicit, bounded bulkheads at the HTTP ingress tier, minimize critical section durations, and isolate foreign functions. SRE teams must abandon legacy `ThreadMXBean` metrics in favor of JFR continuous event streaming and HikariCP connection queue telemetry.

7.5.2 Target Peer-Reviewed Venues

The research contributions possess direct publication potential across two premier international venues:

1. **ACM/SPEC International Conference on Performance Engineering (ICPE):** Target Track: System Performance and Benchmarking Methodologies. Contribution mapping: Empirical decoupling of carrier unpinning from Amdahl single-server serialization ceiling ($\mu = 1/D = 2.0$ req/s), forensic discovery of JNI wait-state diagnostic blind spot, and the dual-window steady-state analysis standard.

2. **IEEE Software:** Target Track: Architecture and Design. Contribution mapping: Three-Tier Concurrency Taxonomy, Five Mandatory Architectural Rules, Four Refactoring Patterns, and Master Decision Tree.

7.5.3 Cumulative Research Protocol: Java 24 GA Replication Roadmap

To enable immediate, independent empirical replication of this thesis's findings on Java 24 GA (JEP 491 [41]):

- **Runtime Configuration:** OpenJDK 24 GA, Apple Silicon host, JVM flags: `-Djdk.virtualThreadScheduler.parallelism=1 -Xms2g -Xmx2g`.

- **Workload:** k6 v1.7.1, 10 closed-loop VUs, think time $Z = 100$ ms, endpoint `/payments` (500 ms serialized mutex).
- **Hypotheses to Replicate:**
  1. *H1 (Unpinning):* `jdk.VirtualThreadPinned` events drop from 108 to strictly 0.
  2. *H2 (Conservation):* Throughput remains clamped to Amdahl ceiling ($X \approx 1.96$ req/s), proving unpinning does not unserialize domain locks.
  3. *H3 (Dispersion):* Evaluate whether JDK 24 `ObjectMonitor` unmounting exhibits lower sample variance than non-fair `ReentrantLock` ($\sigma = 16.53$ ms).

Chapter 8

Conclusion

8.1 Answers and Achieved Goals

This dissertation has conducted a rigorous empirical systems investigation into the architectural, performance, and operational implications of adopting Java 21 Virtual Threads (Project Loom) within enterprise Spring Boot Model-View-Controller (MVC) applications. Through an instrumented Spring Boot 3.5.12 testbed subjected to 125 physical benchmark executions across eight non-overlapping strata, this research evaluated whether user-mode thread multiplexing resolves enterprise concurrency bottlenecks or introduces new failure modes.

The canonical research questions are definitively answered:

- **Answer to RQ1 (Synchronization and Schedulability):** In Java 21 LTS, invoking blocking operations inside intrinsic monitors (`synchronized`) pins virtual threads to carrier threads (108 JFR events), starving single-carrier schedulers. Replacing `synchronized` with `ReentrantLock` completely eliminates carrier pinning (0 JFR events), but yields **identical throughput** (1.956 vs. 1.967 req/s) because Amdahl's Law enforces an immutable serialization ceiling ($\mu = 1/D \approx$ 2.0 req/s). Carrier unpinning restores carrier availability for independent tasks, but cannot unserialize an application critical section. Furthermore, default non-fair `ReentrantLock` direct-CAS barging inflates tail-latency standard deviation by a factor of 23 ($\sigma = 16.53$ ms vs. 0.70 ms). Upgrading to Java 24 (JEP 491 [41]) eliminates monitor pinning at the runtime level (0 events) while throughput remains constrained to the Amdahl ceiling.

- **Answer to RQ2 (Architectural Migration Risks):** Before enabling Virtual Threads, software engineering teams must audit four critical boundaries:

  1. *Foreign Function Interfaces (JNI):*
     Blocking POSIX system calls (`nanosleep()`) capture OS carrier threads in Darwin/Mach wait states ($1.04\% \pm 0.12\%$ process CPU) with **zero JFR pinning events**, creating a severe diagnostic blind spot.
  2. *Relational Persistence Pools (HikariCP):* Virtual threads holding database connections across blocking operations unmount cleanly without pinning carrier threads (0 JFR events), but throughput is strictly governed by Little's Law ($X_{\max} = K/T_{\text{hold}}$). Pool size 1 capped throughput at 3.27 req/s with 49 queued virtual threads, while pool size 10 scaled to 32.60 req/s with 1.43 s median latency, validating the Downstream Bottleneck Shift Law.
  3. *Context Propagation (`ThreadLocal`):* Retaining request context across millions of virtual threads introduces linear $O(N)$ heap scaling.
  4. *Observability Infrastructure:* JFR filters pinning events < 20 ms, while legacy `ThreadMXBean` metrics omit virtual threads entirely.

8.2 Contribution Summary

The primary contributions delivered by this research comprise:

1. **Decoupling Carrier Unpinning from Domain Serialization:** Providing mathematical and empirical proof that eliminating monitor pinning via `ReentrantLock` does not increase throughput for serialized critical sections.
2. **Discovery of Silent JNI Carrier Starvation:** Proving that native C blocking calls capture carrier threads without emitting JFR pinning telemetry, maintaining negligible CPU utilization in kernel wait states.
3. **The Three-Tier Concurrency Taxonomy:** Formalizing the separation of concerns between Tier 1 Schedulability, Tier 2 Domain Synchronization, and Tier 3 Backend Capacity, anchored by three foundational architectural principles.
4. **The Dual-Window Reporting Standard:** Establishing that isolating steady-state hold windows (15–45s) is mandatory to eliminate the 25.8% latency under-reporting distortion induced by warm-up ramps in closed-loop benchmarks.

5. **The Authoritative Engineering Migration Framework:** Formulating five mandatory architectural rules, four correctness-preserving refactorings, and a master decision tree for enterprise migration.

6. **The Open-Science Evidence Freeze Archive:** Preserving an audited repository corpus of 504 physical disk files (966 MB) across 125 executions, verified via SHA-256 cryptographic digests.

8.3 Core Engineering Lessons and Paradigm Shift

The adoption of Virtual Threads in Java 21 LTS represents a profound paradigm shift: threads are transformed from heavyweight, pooled resources into cheap, ephemeral execution units. However, this dissertation proves that **virtual threads are not an architectural panacea**. Virtual threads virtualize execution scheduling; they do not virtualize physical hardware, critical section serialization, or downstream database capacity.

Historically, bounded web-tier thread pools acted as crude admission limiters. Removing the platform thread ceiling transfers concurrency pressure directly to downstream database pools and internal locks. Consequently, the primary responsibility of enterprise software architects transitions from passive thread pool tuning to active architectural capacity management: enforcing perimeter bulkheading, minimizing critical section durations, isolating native foreign calls, and sizing connection pools strictly to physical database capacity.

8.4 Limitations and Boundary Conditions

The findings of this dissertation are bounded by the following explicit conditions:

1. **Single-Node Host Environment:** Primary benchmarks were executed on an 8-core Apple M1 host (macOS Darwin 25.2.0, arm64). Absolute throughput and scheduling latencies reflect this bare-metal platform.

2. **Scheduler Isolation Parameterization:** In-process scheduler constraints (`parallelism=1`) were applied to isolate mechanisms; multi-carrier scaling was evaluated separately in Stratum 5 (8 carriers).

3. **Closed-Loop Workload Regime:** Load generation utilized closed-loop k6 profiles ($N = 10$ or 50 VUs, $Z = 100$ ms), enforcing stationary queueing equilibrium; open-loop burst traffic was not evaluated.

4. **Descriptive Sample Size Bounds:** Benchmark conditions were replicated across $N = 3$ process repetitions, reported strictly as descriptive engineering bounds without inferential over-claiming.

5. **In-Process Database Scope:** Persistence testing evaluated single-node HikariCP pooling; multi-node distributed clustering and external connection proxies (e.g., PgBouncer) were not benchmarked.

8.5 Future Research Directions

Based on the boundaries identified in this research, five prioritized future research trajectories are outlined:

1. **Direction 1: Open-Loop Arrival Workload Modeling.** Evaluating open-loop Poisson arrival traffic ($\lambda > \mu$) to model catastrophic queue growth, connection backlog drops, and client timeout cascades under unbounded virtual thread instantiation [43].

2. **Direction 2: Linux CFS Bandwidth Throttling.** Investigating virtual thread carrier scheduling dynamics under Linux Completely Fair Scheduler (CFS) container quotas (`cpu.cfs_quota_us`) in Kubernetes multi-tenant pods.

3. **Direction 3: Project Panama Foreign Function & Memory (FFM) API Mechanics.** Evaluating whether foreign downcalls executed via Project Panama (JEP 454 [9]) exhibit identical carrier captivity to JNI or permit cooperative runtime unmounting.

4. **Direction 4: Multi-Node Distributed Database Persistence.** Benchmarking virtual thread connection checkout behavior against distributed database clusters with network partition injection and connection pool multiplexers.

5. **Direction 5: Hardware Power Consumption and Energy Efficiency.** Measuring physical processor power draw (watts and joules per request) compar-

ing platform threads, virtual threads, and reactive event loops under equivalent throughput regimes.

Bibliography

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference (AFIPS '67)*, pages 483–485, New York, NY, 1967. ACM. doi: 10.1145/1465482.1465560.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, 1991. doi: 10.1145/146941.144444.

[3] Apache Software Foundation. Apache Tomcat 10.1 HTTP Connector and Thread Pool Architecture (v10.1.52). `https://tomcat.apache.org/tomcat-10.1-doc/`, 2024.

[4] David F. Bacon, Clement R. Attanasio, David F. Lee, V. T. Rajan, and Stephen Smith. Java without the overhead: Thin locks. *ACM SIGPLAN Notices*, 36(5): 63–72, 2001. doi: 10.1145/381694.378814.

[5] Danijel Beronić, Branko Mihaljević, and Aleksander Radovan. On analyzing virtual threads—a structured concurrency model for scalable applications on the JVM. In *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 954–959, Opatija, Croatia, 2021. IEEE. doi: 10.23919/MIPRO52101.2021.9596855.

[6] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asit M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking

development and dataset. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*, pages 169–190, New York, NY, 2006. ACM. doi: 10.1145/1167473.1167488.

[7] Lubomír Bulej, Tomáš Bureš, Jiří Keznikl, Petr Tůma, and Vojtěch Horký. Unit testing performance in java. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*, pages 175–186, New York, NY, 2017. ACM. doi: 10.1145/3030207.3030222.

[8] Daniel Charlak, Jerzy Brzeziński, and Grzegorz Kozieł. Comparative analysis of reactive programming and java virtual threads. *Journal of Computer Sciences Institute*, 38:94–101, 2026. doi: 10.35784/jcsi.9409.

[9] Maurizio Cimadamore and Per-Åke Minborg. JEP 454: Foreign Function & Memory API. `https://openjdk.org/jeps/454`, 2024. Delivered in JDK 22.

[10] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965. doi: 10.1145/365559.365617.

[11] dotCMS Core Engineering Team. Revert virtual threads on asynchronous commit listeners due to lock pinning and carrier starvation. GitHub Pull Request #36900, 2024. URL `https://github.com/dotCMS/core/pull/36900`.

[12] ELTE Faculty of Informatics. Ai usage policy. `https://www.inf.elte.hu/en/ai-usage-policy`, 2026. Accessed 2026-09-11.

[13] ELTE Faculty of Informatics. Guidelines on thesis and final examination: Msc. `https://www.inf.elte.hu/en/guidelines-thesis-final-examination-msc`, 2026. Accessed 2026-09-11.

[14] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '07)*, pages 57–76, New York, NY, 2007. ACM. doi: 10.1145/1297027.1297033.

[15] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, Boston, MA, 2006. ISBN 978-0321349606.

[16] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. *The Java Language Specification, Java SE 21 Edition*. Oracle America, Inc., Santa Clara, CA, 2023. URL `https://docs.oracle.com/javase/specs/jls/se21/html/`.

[17] Grafana Labs. k6 load testing documentation: Closed-loop virtual users, iteration models, metric thresholds, and json lines telemetry (v1.7.1). `https://k6.io/docs/`, 2024.

[18] Andrew Haley and Andrew Dinn. JEP 506: Scoped Values. `https://openjdk.org/jeps/506`, 2025. Planned for JDK 25.

[19] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann / Elsevier, Burlington, MA, 2008. ISBN 978-0123705914.

[20] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974. doi: 10.1145/355620.361161.

[21] Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM '13)*, pages 63–74, New York, NY, 2013. ACM. doi: 10.1145/2555670.2464160.

[22] Leonard Kleinrock. *Queueing Systems, Volume 1: Theory*. John Wiley & Sons, New York, NY, 1975. ISBN 978-0471491101.

[23] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979. doi: 10.1109/TC.1979.1675439.

[24] Luka Lašić, Danijel Beronić, Branko Mihaljević, and Aleksander Radovan. Assessing the efficiency of java virtual threads in database-driven server applications. In *2024 47th International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1475–1480, Opatija, Croatia, 2024. IEEE. doi: 10.1109/MIPRO60963.2024.10569754.

[25] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Professional, Boston, MA, 2nd edition, 2000. ISBN 978-0201310092.

[26] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande (JAVA '00)*, pages 36–43, New York, NY, 2000. ACM. doi: 10.1145/337449.337465.

[27] Paweł Likus and Filip Krużel. Analysis of virtual threads in spring applications. In *Proceedings of the 39th ECMS International Conference on Modelling and Simulation (ECMS 2025)*, pages 555–562. European Council for Modelling and Simulation, 2025. doi: 10.7148/2025-0555.

[28] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java Virtual Machine Specification, Java SE 21 Edition*. Oracle America, Inc., Santa Clara, CA, 2023. URL `https://docs.oracle.com/javase/specs/jvms/se21/html/`.

[29] John D. C. Little. A proof for the queuing formula: $L = \lambda W$. *Operations Research*, 9(3):383–387, 1961. doi: 10.1287/opre.9.3.383.

[30] John D. C. Little. Little's law as viewed by its researcher: History and discussions. *Operations Research*, 59(3):536–549, 2011. doi: 10.1287/opre.1110.0940.

[31] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*, pages 378–391, New York, NY, 2005. ACM. doi: 10.1145/1040305.1040336.

[32] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991. doi: 10.1145/103727.103729.

[33] Gregory Reynolds Morse. Msc thesis supervision guidance. `https://grm.web.elte.hu/`, 2026. Accessed 2026-09-11.

[34] Juan Navarro, Pedro Sánchez, and Francisco Morales. Considerations for integrating virtual threads in a java framework: A quarkus example in a resource-constrained environment. In *Proceedings of the 14th ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion)*, pages 99–104, New York, NY, 2023. ACM. doi: 10.1145/3583678.3596895.

[35] OpenJDK. OpenJDK HotSpot Virtual Machine Source Code (Branch: jdk-21.0.8-ga). https://github.com/openjdk/jdk21u, 2024. Specifically: continuationFreeze.cpp, continuation.cpp, and jvmtiThreadState.cpp.

[36] Oracle Corporation. Virtual threads (java se 21 documentation). https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html, 2023. Oracle Core Libraries Guide.

[37] John Ousterhout. Why threads are a bad idea (for most purposes). In *USENIX Annual Technical Conference, Invited Talk*, San Diego, CA, 1996.

[38] Ron Pressler. JEP 425: Virtual Threads (Preview). https://openjdk.org/jeps/425, 2022. Delivered in JDK 19.

[39] Ron Pressler. JEP 436: Virtual Threads (Second Preview). https://openjdk.org/jeps/436, 2023. Delivered in JDK 20.

[40] Ron Pressler. JEP 444: Virtual Threads. https://openjdk.org/jeps/444, 2023. Delivered in JDK 21 LTS.

[41] Ron Pressler. JEP 491: Synchronize Virtual Threads without Pinning. https://openjdk.org/jeps/491, 2024. Delivered in JDK 24.

[42] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009. doi: 10.1007/s10664-008-9102-8.

[43] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: A cautionary tale. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design & Implementation (NSDI '06)*, pages 239–252, Berkeley, CA, 2006. USENIX Association.

[44] Luca Traini, Vittorio Cortellessa, Daniele Di Pompeo, and Michele Tucci. Towards effective assessment of steady state performance in java software: Are we there yet? *Empirical Software Engineering*, 28(1):13, 2022. doi: 10.1007/s10664-022-10247-x.

[45] VMware / Broadcom. Spring Boot Reference Documentation: Virtual Threads and Web Applications (v3.5.12). https://docs.spring.io/spring-boot/docs/3.5.12/reference/html/, 2024.

[46] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, pages 19–24, Berkeley, CA, 2003. USENIX Association.

[47] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 230–243, New York, NY, 2001. ACM. doi: 10.1145/502034.502057.

[48] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Science & Business Media, Berlin, Heidelberg, 2012. doi: 10.1007/978-3-642-29044-2.

[49] Brett Wooldridge. HikariCP: Downstream Connection Pool Sizing and Optimization Guidelines (v6.3.3). `https://github.com/brettwooldridge/HikariCP/wiki/About-Pool-Sizing`, 2023.

List of Tables

1.1 Research questions, operational deconstruction, and intended evidence. . . 9
1.2 Symptom-to-Mechanism Disambiguation Matrix. 10
1.3 Contribution and responsibility boundary ledger. 13

2.1 Comprehensive literature taxonomy across foundational concurrency domains. 17
2.2 Systematic Comparative Delta Matrix against closest related work. 24

4.1 Research-question-to-method operationalisation matrix. 39
4.2 Master repository execution census across eight evidence strata. 41
4.3 Methodological threats to validity and experimental mitigations. 44

5.1 Empirical endpoint catalog and experimental roles. 46
5.2 Tomcat request dispatch architecture: Virtual vs. Platform concurrency. . . 47

6.1 Master result provenance ledger mapping claims to empirical artifacts. . . . 53
6.2 Java 21 primary synchronization matrix performance (Stratum 1). 54
6.3 Steady-state trial-level dispersion across $N = 3$ process repetitions. 55
6.4 Java 21 vs. Java 24 monitor synchronization performance (Stratum 2). . . 56
6.5 JNI foreign function execution telemetry (Stratum 3, $N = 9$). 56
6.6 Two-Treatment Factorial Persistence Ablation performance (Stratum 8). . 57
6.7 Supporting architectural diagnostic controls (Stratum 4). 57
6.8 Systematic negative, null, and ablation findings ledger. 58

7.1 Research-question closure ledger. 60
7.2 Architectural migration triage and action matrix. 65
7.3 Master threats to validity matrix. 66

A.1 Reproduction and independent-check manifest. 82

B.1 Research governance and disclosure record. 86

B.2 Detailed inventory of generative AI tool usage (Dean's Directive 1/2026 Annex 3). 87

Appendix A

Reproduction and Independent-Check Record

A.1 Reproduction Overview and Open-Science Package

In accordance with open-science standards and ELTE Faculty of Informatics thesis requirements, the complete empirical corpus supporting this dissertation has been frozen, audited, and cataloged within the repository under `docs/evidence-freeze/`. The package encompasses 504 physical disk files totaling 965.99 MB, including raw k6 JSON lines streams, binary Java Flight Recorder (JFR) event files, process telemetry CSV logs, and execution manifests.

Table A.1 presents the complete Reproduction and Independent-Check Manifest, providing exact values, file references, and deterministic commands to independently check all representative findings.

Table A.1: Reproduction and independent-check manifest.

Item	**Exact Value or Procedure**	**Expected Evidence**
Frozen Revision	Git commit hashes `1ccb617b6beecb25` `0e534d758d4f0eb1ea67e2d2` and `5f9936c`; frozen directory `docs/evidence-freeze/`.	Commit hashes recorded in 113 JSON execution manifests.

Item	Exact Value or Procedure	Expected Evidence
Environment	Hardware: Apple M1 (8 cores: 4P+4E), 8 GB RAM. OS: macOS Darwin 25.2.0 (arm64). Runtime: OpenJDK 21.0.8, OpenJDK 24.0.2, Maven 3.9, k6 v1.7.1, Spring Boot 3.5.12.	Verified in `run-family-matrix.csv` across 125 physical trials.
Inputs	504 cataloged physical disk files (966 MB) partitioned across 8 strata, with cryptographic SHA-256 digests in `artifact-inventory.csv`.	100% checksum match verified across 368 result/JFR and 92 telemetry files.
Automated Tests	`rtk mvn test`	8 test suites run, 0 failures, 0 errors, BUILD SUCCESS (verifies locks, JNI, JDBC).
Evidence Audit	`rtk python3 scripts/validate_normalized_evidence.py`	76 normalized JSON files verified with 100% SHA-256 integrity match.
Metric Recalculation	`rtk python3 scripts/recalculate_metrics.py--verify-reconciliation`	Parses 150,000+ k6 JSON lines; reconciles steady-state vs. whole-run delta (0.765%).
TDK Evaluation Audit	`rtk python3 ELTE_IK_TDK_50_Point_Audit_Universal/elte-ik-tdk-50-point-audit/scripts/score_tdk.py docs/evidence-freeze/review-tdk-50point.json`	Evaluates 50-point criteria score based on ratified audit records.

Item	Exact Value or Procedure	Expected Evidence
Central Result: Mutex	Stratum 1: `payments-synchronized` vs. `payments-reentrant-lock` under single carrier.	108 pinning events reduced to 0; throughput identical at 1.956 vs. 1.967 req/s.
Central Result: JNI	Stratum 3: `blocking-jni` (500 ms POSIX `nanosleep()`).	Throughput collapses to 1.967 req/s; 0 JFR pinning events; process CPU $1.04\% \pm 0.12\%$.
Central Result: Pool	Stratum 8: Two-Treatment Factorial Persistence Ablation.	Pool 1: 3.27 req/s ($p50 \approx 15.1$ s, 49 queued VUs); Pool 10: 32.60 req/s ($p50 \approx 1.43$ s).
Known Limits	In-process single-carrier constraint (`parallelism=1`); closed-loop 10/50 VUs; $N = 3$ process repetitions reported as descriptive bounds.	Multi-carrier baseline documented in Stratum 5; inferential ANOVA excluded.

Appendix B

Ethics, Data, Licensing, and AI Record

B.1 Research Governance and Compliance Overview

This dissertation strictly adheres to the research governance policies of the Faculty of Informatics, Eötvös Loránd University (ELTE), including the Academic Regulations for Students (Volume 2, Sections 74/A–E) and **Dean's Directive No. 1/2026 (III. 26.)** on the use of Artificial Intelligence tools in higher education, research, and thesis preparation [12].

Table B.1 summarizes the research governance, data licensing, ethical boundaries, and AI disclosure record.

Table B.1: Research governance and disclosure record.

Governance Area	Status and Authority	Evidence Location	Residual Risk or Action
Ethics and Consent	Not applicable (systems engineering study).	No human subjects, personal data, or clinical interventions involved.	None; zero personal or sensitive user data processed.
Privacy and Confidentiality	Fully compliant; public open-science artifact.	All source code and telemetry are public; zero corporate credentials or private keys.	All runtime passwords in scripts are local test dummies (e.g., `postgres:thesis`).
Data and Artifact Licences	Open-source distribution under the MIT License.	`LICENSE` file in repository root; evaluated dependencies are Apache 2.0, MIT, or EPL-2.0.	Full redistribution rights granted for open-science replication.
AI Assistance	Declared under Dean's Directive 1/2026 (Annex 3).	Table B.2; signed declaration filed with thesis submission.	Zero AI-generated data; all empirical data originates from physical hardware executions.

B.2 Detailed Disclosure of Generative AI Assistance

In accordance with Annex 3 of Dean's Directive 1/2026, Table B.2 provides a detailed accounting of all generative AI tool usage during the preparation of this thesis.

Table B.2: Detailed inventory of generative AI tool usage (Dean's Directive 1/2026 Annex 3).

Purpose	**Method and Critical Adaptation**	**AI Tool Used**	**Thesis Scope / Location**
Structural Drafting, Outlining, and Prose Polish	Generated preliminary outlines and structural drafts for standard framework descriptions; critically revised and rewritten by author to align with ELTE academic standards and queueing models.	Anthropic Claude (Claude 3.5 Sonnet)	Chapters 1 (§1.1), 3 (§3.1), 4 (§4.1), 5 (§5.1), 7 (§7.1), and 8.
Scaffolding Recalculation & Verification Scripts	Scaffolding preliminary Python templates for parsing raw k6 JSON lines and JFR events; audited, refined, and verified against physical data using cryptographic SHA-256 checks.	Anthropic Claude (Claude 3.5 Sonnet) & GitHub Copilot	Python verification scripts (`scripts/normalize_evidence.py`, `recalculate_metrics.py`, `validate_normalized_evidence.py`).
Integration Test Boilerplate Scaffolding	Autocompleted boilerplate JUnit 5 integration test fixtures; manually adapted with strict assertions for mutual exclusion, `ThreadLocal` cleanup, and JNI bindings.	GitHub Copilot	Chapter 5 and test directory `src/test/java/`.

B.2.1 Strict Exclusions and Independent Authorship Guarantees

1. **Zero AI Generation of Empirical Data:** No benchmark results, throughput metrics, latency distributions, JFR pinning events, or host CPU telemetry were generated, simulated, or interpolated by artificial intelligence. Every data point reported in this dissertation originates exclusively from physical benchmark executions on dedicated bare-metal hardware.

2. **Independent Theoretical Attribution:** All theoretical derivations (e.g., Amdahl single-server saturation, Little's Law response-time calculations, and Downstream

Bottleneck Shift Law), research question closures, and the Three-Tier Concurrency Taxonomy reflect the author's independent intellectual analysis.

3. **Primary Source Literature Audits:** All cited literature, OpenJDK JEP specifications, and HotSpot C++ runtime source implementations (`continuationFreeze.cpp`, `VirtualThread.java`) were independently verified against primary authoritative sources.

Appendix C

Process and Submission Checklist

C.1 Master Regulatory and Submission Verification

In accordance with the MSc Thesis Regulations of the Faculty of Informatics, Eötvös Loránd University (ELTE), Dean's Directive No. 1/2026, and the official thesis evaluation criteria, this checklist documents that all twelve administrative, structural, methodological, and regulatory requirements have been verified prior to final submission.

[✓] **Administrative Consistency:** Approved topic registration (`IK-ZVMSC_EN2_155`), degree programme (Computer Science MSc — Software Architecture Specialisation), thesis title, candidate metadata (Hamed Mohamed, `DG1EWF`), supervisory team (Dr. habil. Tamás Kozsik and Gregory Reynolds Morse), and submitted content are strictly consistent.

[✓] **Goal and Question Closure:** Every approved research goal and canonical question (RQ1 and RQ2) closes with direct empirical evidence, including negative and null findings (Table 7.1 and Table 6.8).

[✓] **Related Work Synthesis:** Chapter 2 verifies and synthesizes theoretical foundations, closest prior work, competing approaches, and establishes the 10-point Comparative Delta Matrix (Table 2.2).

[✓] **Result Provenance Traceability:** Every central result traces to a fixed method, input dataset, software artifact version, and cryptographic SHA-256 manifest entry (Table 6.1 and Table A.1).

[✓] **Student Contribution Boundaries:** Student independent contribution and responsibility boundaries are explicit and consistent across the thesis text, TDK statement, and repository ledger (Table 1.3).

[✓] **Document Integrity:** Thesis PDF and editable LaTeX source derive from the identical frozen repository revision; all internal hyperlinks, citations, captions, lists, and cross-references resolve without broken targets.

[✓] **ELTE Formatting Compliance:** Document formatting strictly complies with ELTE IK standards: A4 paper, 12-point body text, 1.5 MS Word equivalent line spacing (`\setstretch{1.427465}`), 25 mm uniform margins, continuous Arabic page numbering, new-page chapter starts, and 50 substantive pages.

[✓] **Final Examination Integration:** The three registered final-examination subjects—*Project Management*, *Advanced Java programming*, and *Scalable enterprise applications*—are naturally integrated throughout the architectural and systems chapters without artificial headings.

[✓] **Submission Package Completeness:** The digital submission package contains the final thesis PDF, approved topic registration form (`approved-topic-registration.pdf`), complete evidence freeze (`docs/evidence-freeze/`), full source code, and signed AI tool declaration.

[✓] **Confidentiality and Data Hygiene:** No corporate secrets, personal student data, private credentials, or unlicensed confidential materials are included in the repository or thesis (Table B.1).

[✓] **Supervisory Evidence:** Documented supervisory feedback, milestone progress reports, consultation slide decks, and completed thesis pre-defense presentations are archived in the project repository.

[✓] **Defense Narrative Alignment:** The thesis defense slide deck and presentation narrative emphasize research questions, empirical systems methodology, quantitative results, and architectural contributions rather than a mechanical software feature walkthrough.