Client Archetypes

&

Rust Strategy Advisory

Contents

[1. Introduction 4](#_Toc197696026)

[2. Client Archetypes Overview 4](#_Toc197696027)

[2.1 Internal Tooling Developers 4](#_Toc197696028)

[🔸 Consultant Feedback 4](#_Toc197696029)

[🔸 Benchmark Mapping 5](#_Toc197696030)

[🔸 Strategic Fit Summary 5](#_Toc197696031)

[🔸 Limitations & Open Questions 5](#_Toc197696032)

[2.2 Monolith Extension Teams 5](#_Toc197696033)

[🔸 Consultant Feedback 5](#_Toc197696034)

[🔸 Benchmark Mapping 5](#_Toc197696035)

[🔸 Strategic Fit Summary 6](#_Toc197696036)

[🔸 Limitations & Open Questions 6](#_Toc197696037)

[2.3 API Backend Teams 6](#_Toc197696038)

[🔸 Consultant Feedback 6](#_Toc197696039)

[🔸 Benchmark Mapping 7](#_Toc197696040)

[🔸 Strategic Fit Summary 7](#_Toc197696041)

[🔸 Limitations & Open Questions 7](#_Toc197696042)

[3. Tool Fit & Benchmark Mapping 7](#_Toc197696043)

[3.1 Code Analysis: 8](#_Toc197696044)

[🔹 Manual CSV Processor 8](#_Toc197696045)

[🔹 Batch Age Group Counter 8](#_Toc197696046)

[🔹 Parallel Batch Group Counter 9](#_Toc197696047)

[🔹 API Backend Teams *(Future Scope)* 10](#_Toc197696048)

[Conclusion 10](#_Toc197696049)

[4. Migration Strategy Recommendations 11](#_Toc197696050)

[4.1 Strategic Migration Criteria. 11](#_Toc197696051)

[1. Performance Gain 11](#_Toc197696052)

[2. Code Isolation and Ownership 11](#_Toc197696053)

[3. Integration Risk 12](#_Toc197696054)

[4. Developer Readiness 12](#_Toc197696055)

[5. Operational Fit 12](#_Toc197696056)

[Summary 13](#_Toc197696057)

[4.2 Tool-by-Tool Strategy Overview 13](#_Toc197696058)

[Tool Migration Table 13](#_Toc197696059)

[4.3 Migration Phase Timeline 15](#_Toc197696060)

[Migration Timeline Table 15](#_Toc197696061)

[4.4 Technical Considerations (real-world blockers) 15](#_Toc197696062)

[4.5 Organizational Risk & Cost Commentary (non-technical side) 15](#_Toc197696063)

[4.6 Recommendation Summary 15](#_Toc197696064)

[Final Recommendations: 15](#_Toc197696065)

[4.7 What We Learned 16](#_Toc197696066)

[Conclusion 16](#_Toc197696067)

1. Introduction

Many backend teams still rely on C# for CLI and batch processing tools that are slow, memory-heavy, and difficult to maintain. This advisory proposes a strategic shift to Rust for these domains. Benchmarked on real-world tasks, Rust tools executed up to 3.6× faster, used 4.7× less memory, and consumed a fraction of CPU time compared to C#. This document targets three specific client types where these improvements provide the most value. API services are excluded from the current phase due to architectural scope but remain candidates for future benchmarking.

# 2. Client Archetypes Overview

This chapter identifies the three client archetypes that Rust CLI and batch tooling directly supports. Each represents a distinct development environment with known performance and reliability challenges. These are the groups where Rust adoption provides the highest return:

* **Internal Tooling Developers**
  + Build and run CLI tools used in CI/CD. They face memory leaks, slow startup times, and runtime failures tied to .NET GC and event mismanagement.
* **Monolith Extension Teams**
  + Patch legacy systems using external batch logic. They avoid touching core code because C# changes cause regressions. Their batch jobs often run slowly and can’t scale.
* **API Backend Teams** 
  + Manage async services under tight latency. They struggle with GC pauses and memory pressure from high-throughput workloads. Rust’s async model offers future promise, but this is untested.

## **2.1 Internal Tooling Developers**

Internal tooling developers maintain command-line utilities, processing scripts, and CI/CD automation steps. These tools are often triggered in headless environments such as build agents, internal schedulers, or deployment pipelines. Their execution must be reliable, memory-efficient, and fast. Most of these tools in practice are written in C#, using basic I/O, LINQ queries, and ad-hoc error handling. As codebases grow or input sizes increase, problems such as memory leaks, sluggish startup, and unpredictable runtime behavior emerge.

### 🔸 Consultant Feedback

A C# consultant (con-1) confirmed that internal tools frequently suffer from static memory leaks (e.g., unhooked event handlers), excessive memory usage, and slowness caused by large codebase load times. These issues persist even in cases where the tool’s business logic is simple — suggesting that the problem is in the runtime behavior, not developer skill or design.

### 🔸 Benchmark Mapping

The Rust-based manual CSV processor directly addresses these pain points. It filters and transforms large CSV files using stream-based processing and compile-time guarantees. In benchmark tests, this tool completed the job **1.7× faster**, with **4.7× less peak memory** and **3× less CPU time** than its C# equivalent. These results confirm that Rust’s lightweight runtime and memory safety model eliminate the exact performance risks the consultant described.

### 🔸 Strategic Fit Summary

Internal tooling is the **highest-priority migration target**. Rust offers immediate, measurable improvements in performance and resource efficiency, while CLI tools can be migrated incrementally without touching upstream infrastructure. These tools are also easier to test, deploy, and wrap inside existing DevOps pipelines.

### 🔸 Limitations & Open Questions

The only constraint is Rust developer onboarding. Teams may need guidance on error handling, build tooling, and packaging practices. However, once set up, Rust’s long-term maintainability and safety offset the upfront friction. No significant obstacles were found that would block partial or full migration of internal tooling scripts.

## 2.2 Monolith Extension Teams

Monolith extension teams maintain large, business-critical systems that cannot tolerate structural change. These systems — such as ERP platforms, billing engines, or internal batch pipelines — are often fragile, tightly coupled, and historically unstable under modification. To avoid introducing regressions or breaking production logic, these teams offload functionality through scheduled tasks or external executables, often written in C#.

### 🔸 Consultant Feedback

A senior C# consultant (con-1) confirmed that developers often intentionally avoid optimizing known inefficient components within the base system. Historical changes to these areas have led to production failures or data corruption, forcing teams to abandon improvements to preserve stability. This reinforces the need for **safe isolation** of new logic outside the monolith boundary.

“There are legacy parts that no one wants to touch anymore… They work, but they’re slow and not scalable. Optimizing them just breaks something else.” (con-1)

### 🔸 Benchmark Mapping

The Rust-based batch and parallel job processors align perfectly with this constraint. These tools were built as external .exe programs, handling CSV grouping and aggregation without needing to touch the C# monolith. Benchmarks show the Rust batch job ran **3.6× faster** than C#, with **similar memory use** but **dramatically lower CPU time**. The parallel Rust version maintained the same memory efficiency and improved runtime by an additional 1.5–2×.

These results demonstrate that Rust can replace or wrap monolithic data processing safely, predictably, and with significant performance gain — without changing fragile base code.

### 🔸 Strategic Fit Summary

Monolith extensions are a **high-value, medium-risk migration target**. Rust tools can act as drop-in replacements for batch scripts or reporting jobs that interact with legacy databases or files. This enables performance and concurrency improvements while avoiding direct interference with the monolith. However, deployment may require closer coordination with system owners and legacy constraints.

### 🔸 Limitations & Open Questions

The primary limitation is integration complexity. Rust tooling works best when the monolith is already designed to invoke external processes. If legacy systems lack this flexibility, additional effort is needed to insert and orchestrate Rust binaries. No functional issues were found during benchmarked use, but operational rollout remains a constraint to validate in production environments.

## 2.3 API Backend Teams

API backend teams build asynchronous services such as REST APIs, webhook processors, or event-driven consumers. These systems operate under strict latency, concurrency, and throughput constraints, typically implemented in modern C# with async/await, task scheduling, and background queuing. Their technical environment is highly dynamic and requires fine-grained control over memory, error handling, and response time under varying load.

Although this archetype was not targeted by implemented benchmarks, it remains strategically relevant due to its reliance on runtime behavior and GC-sensitive performance characteristics.

### 🔸 Consultant Feedback

The consultant (con-1) reported **difficulty managing memory behavior** under load and acknowledged that **leaks or static event binding issues** continue to degrade reliability. While these problems were highlighted in the context of CLI tooling, they are **even more impactful** in high-concurrency API services, where latency budgets are tighter and runtime pauses can affect real users.

“When you’re not careful with events or static stuff, memory spikes or strange crashes just happen. You feel it more when there’s high load.” (con-1)

### 🔸 Benchmark Mapping

No Rust benchmarks were performed for async services. However, based on Rust’s core strengths — no garbage collector, deterministic memory ownership, and modern async runtimes (tokio, actix) — its **theoretical advantage in latency-critical, concurrent services is strong**. If the memory behavior issues observed in CLI tools also apply to async workloads, Rust would likely outperform C# in predictability and baseline resource usage.

### 🔸 Strategic Fit Summary

API backend teams are a **low-priority migration target for this phase**, but a **high-value target for future exploration**. They are currently excluded due to project scope and technical complexity in benchmarking full async services fairly across runtimes. However, the memory and GC concerns raised in consultant feedback indicate a long-term opportunity.

### 🔸 Limitations & Open Questions

No implementation was done in this area, so all conclusions remain theoretical. Rust’s async ecosystem requires more initial setup, and integration with .NET ecosystems (e.g., identity/auth, DB abstraction layers) may present early friction. Benchmarking Rust async APIs vs. C# would require a dedicated effort, including async-safe workload design and service-layer profiling.

3. Tool Fit & Benchmark Mapping

This section shows how each Rust and C# tool was implemented, what performance differences emerged, and why. Each tool is tied directly to a client use case. Rust consistently outperformed C# due to better memory control, safer concurrency, and zero garbage collection. These results validate targeted migration actions for each archetype based on measurable technical impact.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Client Archetype** | **Tool** | **File(s)** | **Rust Advantage** | **Recommended Strategy** | **Priority** |
| Internal Tooling Developers | Manual CSV Processor | cli\_tool.rs / CsvProcessorManual.cs | 1.7× faster, 4.7× less RAM, 3× lower CPU | Replace C# CLI scripts with compiled Rust binaries | High |
| Monolith Extension Teams | Batch Age Group Counter | batch\_job.rs /  BatchJob.cs | 3.6× faster, same RAM, 3× lower CPU | Offload core logic into Rust batch wrappers | Medium |
| Monolith Extension Teams | Parallel Batch Group Counter | batch\_job\_parallel.rs BatchJobParallel.cs | 2.5× faster, 7× less RAM, 4× lower CPU | Use Rust CLI tools for scalable, safe parallelism | Medium |
| API Backend Teams | *(Not implemented)* | — | Predictable async behavior (theoretical only) | Explore Rust async runtimes in a future implementation | Low |

Rust tools consistently addressed real issues found in C#: memory spikes from garbage collection, poor concurrency scalability, and excessive CPU usage from managed allocations. Each tool is tied to a measurable pain point observed by developers and validated through benchmarked outcomes.

## 3.1 Code Analysis:

### 🔹 Manual CSV Processor

**Files:** cli\_tool.rs / CsvProcessorManual.cs  
**Client Fit:** Internal Tooling Developers

**⚙️ Task:**  
Stream a large CSV file, filter users with age > 30, transform names, write output.

**🛠 Implementation Differences:**

* **C#:** Uses StreamReader + Split(','). This approach allocates new string arrays for every row, leading to memory fragmentation and frequent GC.
* **Rust:** Uses csv + serde to deserialize rows into struct types efficiently, avoiding runtime memory churn.

**📊 Benchmark Result:**  
Rust ran **1.7× faster**, used **4.7× less memory**, and burned **3× less CPU**.

**❌ Why C# Failed:**  
C# performs redundant string allocations and depends on GC cleanup. The .Split() method breaks on malformed input and can’t guarantee memory stability during batch runs.

**📈 Migration Strategy:**  
Rust binaries should replace CLI scripts immediately. They eliminate GC risk and scale cleanly in automated pipelines.

### 🔹 Batch Age Group Counter

**Files:** batch\_job.rs / BatchJob.cs  
**Client Fit:** Monolith Extension Teams

**⚙️ Task:**  
Load CSV into memory, group users by age range, count totals, write summary.

**🛠 Implementation Differences:**

* **C#:** Uses CsvHelper and Dictionary<string, int>. Internally boxes values and triggers GC during resizing.
* **Rust:** Uses csv and BTreeMap, which allocates in-place and keeps memory predictable.

**📊 Benchmark Result:**  
Rust was **3.6× faster**, used similar memory, and consumed **3× less CPU**.

**❌ Why C# Failed:**  
C# relies on runtime-allocated containers and boxed operations. GC interrupts mid-aggregation, and boxed values waste cache space.

**📈 Migration Strategy:**  
Use Rust batch wrappers to move critical logic outside the monolith. This reduces CPU pressure without rewriting legacy internals.

### 🔹 Parallel Batch Group Counter

**Files:** batch\_job\_parallel.rs / BatchJobParallel.cs  
**Client Fit:** Monolith Extension Teams (scalable workloads)

**⚙️ Task:**  
Same grouping as above, executed in parallel.

**🛠 Implementation Differences:**

* **C#:** Uses Parallel.ForEach and ConcurrentDictionary. Lock contention and context switching degrade scalability.
* **Rust:** Uses rayon and reduce to parallelize safely, without locks or shared memory overhead.

**📊 Benchmark Result:**  
Rust was **2.5× faster**, used **7× less memory**, and had **4× lower CPU time**.

**❌ Why C# Failed:**  
C# concurrency is GC-bound and requires synchronization. Parallel threads compete for memory and lock access, causing jitter and memory bloat.

**📈 Migration Strategy:**  
For high-volume jobs, Rust parallel tools scale with less risk and better performance. Integrate where C# parallelism plateaus.

### 🔹 API Backend Teams (Future Scope)

**Files:** (No benchmarked tool)  
**Client Fit:** API Backend Teams

**⚙️ Task:**  
Not implemented.

## Conclusion

The benchmarks confirm that Rust delivers consistent, measurable improvements over C# across all tested tools. Each Rust implementation reduced execution time, memory use, and CPU load without sacrificing safety or output accuracy. These gains are not theoretical — they directly align with specific problems reported by developers working on internal scripts, fragile batch jobs, and concurrency-heavy pipelines.

The mapping shows exactly where Rust can replace or wrap existing tools, and what kind of return each client archetype can expect. Internal tooling offers immediate migration wins with low effort. Batch jobs benefit from externalization and safe parallelism. API services remain a future target, but Rust’s async model shows strong potential.

This chapter translates benchmarks into action. It provides the technical foundation for the migration strategies presented in the next section.

# 4. Migration Strategy Recommendations

This chapter maps each benchmarked Rust tool to a concrete migration action. It shows which tools should be replaced, deferred, or phased in based on performance gains, risk level, team ownership, and system constraints. We focus on tools that solve real problems—GC instability, memory bloat, and concurrency failure—and prioritize those that can be deployed without disrupting core systems. The goal is to enable fast, safe adoption of Rust where it delivers immediate value, starting with internal CLI scripts and progressing toward batch integration.

## 4.1 Strategic Migration Criteria.

This section defines the evaluation logic used to prioritize Rust migration targets. These criteria ensure that tool selection isn’t based on benchmark results alone, but on real-world fit, system risk, and team capacity. Each criterion is derived directly from project goals and stakeholder realities. Together, they form a repeatable, evidence-based framework for migration decisions.

### 1. Performance Gain

Measures how much Rust improves execution time, memory usage, and CPU efficiency compared to C#.  
We prioritize tools where performance gaps are significant and directly impact runtime stability, resource cost, or responsiveness.

|  |  |  |
| --- | --- | --- |
| **Aspect** | **⚙️ C#** | **🦀 Rust** |
| 🔼 Strengths | JIT compiles to optimized native code over time (with warmup), easy async handling | Predictable performance, compiled ahead-of-time, no GC, high cache efficiency |
| 🔽 Weaknesses | GC causes jitter, high memory overhead under load | Compile times are longer, and memory safety checks can slow dev speed |

**Why it matters here:** Tools with >2× performance gains (e.g. parallel batch jobs) offer immediate operational ROI and justify early replacement.

### 2. Code Isolation and Ownership

Assesses whether the tool is modular and controlled by a single team without dependencies on shared or fragile systems. Tools that run as standalone binaries or CI scripts score high; embedded or cross-cutting logic scores low.

|  |  |  |
| --- | --- | --- |
| **Aspect** | **⚙️ C#** | **🦀 Rust** |
| 🔼 Strengths | Widely used inside monoliths, deeply integrated with .NET systems | Ideal for standalone tools, can run as external binaries, avoids dependency hell |
| 🔽 Weaknesses | Code is often deeply coupled to shared business logic, making it risky to extract | Poor fit if integration requires direct in-process replacement or runtime reflection |

**Why it matters here:** Internal CLI tools are fully owned and easy to replace. Monolith jobs require coordination and are risk-prone.

### 3. Integration Risk

Evaluates how safely a Rust tool can be introduced without breaking surrounding systems. We consider data I/O formats, legacy dependencies, runtime orchestration, and scripting compatibility.

|  |  |  |
| --- | --- | --- |
| **Aspect** | **⚙️ C#** | **🦀 Rust** |
| 🔼 Strengths | Integrates cleanly with other .NET components, rich ecosystem support | Excellent at writing external command-line tools that communicate via files or pipes |
| 🔽 Weaknesses | Hard to isolate logic in monoliths, changes ripple system-wide | Poor interop with .NET, tricky to embed, requires strict input/output control to integrate safely |

**Why it matters here:** Batch jobs written in Rust can replace .exe tasks cleanly. Embedded C# logic that touches shared memory or auth layers creates high-risk points.

### 4. Developer Readiness

Measures whether the owning team has the Rust knowledge and operational control to maintain the tool after migration. Teams familiar with shell, CLI workflows, and automated builds are better positioned to adopt Rust.

|  |  |  |
| --- | --- | --- |
| **Aspect** | **⚙️ C#** | **🦀 Rust** |
| 🔼 Strengths | Most enterprise teams know C#, tools like LINQ and Visual Studio accelerate dev | Excellent compile-time checks, memory safety, concurrency guarantees once learned |
| 🔽 Weaknesses | Safety is reliant on runtime (GC, null checks), unsafe concurrency possible | Steep learning curve, frustrating compiler errors, slower iteration speed early on |

**Why it matters here:** Even a perfectly written Rust tool is a liability if no one can debug it when it breaks.

### 5. Operational Fit

Determines whether the tool’s runtime behavior aligns with Rust’s architectural strengths—static binaries, predictable memory, and no runtime reflection. Batch-oriented, stateless tools with stable inputs are a strong match.

|  |  |  |
| --- | --- | --- |
| **Aspect** | **⚙️ C#** | **🦀 Rust** |
| 🔼 Strengths | Dynamic config, dependency injection, runtime logging, service control | Fast startup, stable memory profile, safe file I/O, great for stream/batch workloads |
| 🔽 Weaknesses | High memory use over time, unpredictable GC pauses | Poor support for dynamic reloading, verbose config handling, no reflection |

**Why it matters here:** CLI tools and file-driven batch jobs fit Rust’s model perfectly. Dynamic or hot-reloaded services do not.

These criteria are applied tool-by-tool in the following section to recommend where Rust can deliver fast, safe, and cost-effective value—starting with internal CLI scripts and progressing toward parallel batch workloads.

### Summary

|  |  |  |
| --- | --- | --- |
| **Criterion** | **Rust Wins When...** | **C# Holds Advantage When...** |
| Performance Gain | Speed, memory, CPU matter most | Warmed-up JIT or async convenience matters more |
| Code Isolation | Tool is standalone or CI-triggered | Tool is deeply coupled to system state |
| Integration Risk | Tool can run as .exe, via pipe or file | Tool relies on internal .NET interfaces |
| Developer Readiness | Team owns and maintains the tool | Team has no Rust experience or tooling setup |
| Operational Fit | Tool has stable inputs, no dynamic config | Tool needs runtime config, reflection, live reloads |

## **4.2 Tool-by-Tool Strategy Overview**

This section applies the migration criteria from 4.1 to each benchmarked tool, evaluating its readiness, risk level, and strategic value. The goal is to determine which tools can be migrated immediately, which should be phased, and which require further validation. Each recommendation is grounded in performance results, integration feasibility, and team capability.

### Tool Migration Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Tool** | **Client Fit** | **Recommended Action** | **Priority** | **Risk** | **Conditions** |
| Manual CSV Processor | Internal Tooling Developers | Full replacement with Rust | High | Low | Team has CLI ownership + CI access |
| Batch Age Group Counter | Monolith Extension Teams | Replace with Rust batch wrapper | Medium | Moderate | Stable input format, external trigger |
| Parallel Batch Processor | Monolith Extension Teams | Gradual replacement with Rust | Medium | Moderate-High | Requires multi-thread test environment |
| API Services (Theoretical) | API Backend Teams | Defer | Low | High | Async benchmarking not implemented |

#### 🔹 Manual CSV Processor

**Recommended Action:** Replace current CLI script with compiled Rust binary

(cli\_tool.rs / CsvProcessorManual.cs)

**Why:**

* Rust showed **1.7× faster**, **4.7× less RAM**, **3× less CPU**
* Fully isolated tool, no shared dependencies
* Maintained by a small DevOps team with full deployment control

**Risks:** None identified.  
**Conditions:** Rust tool must support same input/output contracts and be integrated into current job scheduler.

#### 🔹 Batch Age Group Counter

**Recommended Action:** Replace existing .NET batch logic with a Rust .exe wrapper (batch\_job.rs)  
**Why:**

* Rust ran **3.6× faster**, reduced CPU by **3×**
* Batch job runs as a scheduled task, uses file I/O (not DB-bound)
* Monolith integration can be avoided with external orchestration

**Risks:** Requires clear ownership boundaries to prevent logic conflicts.  
**Conditions:** File format stability and consistent runtime environment must be confirmed.

#### 🔹 Parallel Batch Group Counter

**Recommended Action:** Phase in Rust parallel implementation (batch\_job\_parallel.rs)  
**Why:**

* Rust ran **2.5× faster**, used **7× less memory**, and required **4× less CPU**
* C# parallel logic suffers from contention via ConcurrentDictionary
* Rust offers lock-free merging via rayon

**Risks:** Higher system impact. Requires thread-safe testing and environment consistency.  
**Conditions:** Teams must validate against real-world data sizes under peak load.

## 4.3 Migration Phase Timeline

This section outlines the recommended rollout order for Rust-based tool migration. The sequence reflects both the technical readiness and the organizational risk level of each tool. The goal is to migrate high-impact, low-risk tools first, then phase in more complex cases after validation.

### Migration Timeline Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Phase** | **Tool** | **Action** | **Why Now** | **Preconditions** |
| Phase 1 | Manual CSV Processor | Immediate full replacement with Rust | Highest benchmark gain, fully isolated, CI/CD owned | Team has control over deployment and format |
| Phase 2 | Batch Age Group Counter | Replace with Rust .exe wrapper | Moderate risk, clear CPU benefit, file-based input/output | Input format stable, job scheduler access |
| Phase 3 | Parallel Batch Processor | Phase-in with test coverage | High performance, but requires concurrency validation | Multithreaded test env, large input scenarios |
| Phase 4 | API Backend Prototype (opt.) | Defer until prototype is benchmarked | No tool exists, high risk, async Rust not validated | Requires future benchmarking and team support |

## 4.4 Technical Considerations (real-world blockers)

Will be updated.

## 4.5 Organizational Risk & Cost Commentary (non-technical side)

Will be updated.

## **4.6 Recommendation Summary**

This migration strategy recommends a phased adoption of Rust-based tools across internal CLI and batch processing workloads. The recommendation is based on benchmarked performance improvements, client-specific constraints, integration feasibility, and developer ownership. Each tool was assessed using consistent migration criteria, and prioritized according to risk, resource fit, and technical return.

### Final Recommendations:

* **Migrate the Manual CSV Processor immediately.** 
  + It is fully isolated, benchmarked, and owned by the CI team. Rust delivers significant performance gains with minimal integration effort.
* ⏳ **Phase in the Batch Age Group Counter next.**
  + It benefits from clear CPU reduction but requires validation of data formats and scheduling control.
* ⏳ **Deploy the Parallel Batch Processor cautiously.**
  + It offers the highest performance gain but requires multithread testing and validation in realistic batch environments.
* ❌ **Defer migration of API backend services.**
  + These are currently unbenchmarked and carry high integration risk. Rust async runtimes should be prototyped and evaluated in a future sprint.

## 4.7 What We Learned

This project confirmed that Rust delivers measurable performance advantages over C# in backend CLI and batch workloads—but also revealed that raw performance isn’t enough. We learned that tools must be selected not only for technical gain, but for how safely they fit into real-world teams, pipelines, and system constraints.

The CLI tool, for example, was replaced easily due to its isolation and DevOps ownership. Batch tools, while showing greater performance gains, exposed integration risks that demand phased rollout and stricter testing. The parallel processor showed the clearest technical win, but also raised the highest operational risks. The API tool space remains promising but unvalidated, and we learned to defer it until we have functional benchmarks in that domain. Consultant feedback helped confirm that memory issues and GC instability are real-world pain points—not just theoretical problems. This reinforced the need to focus on tools already causing friction in existing workflows.

Going forward, we will continue to follow the full design cycle: define a target problem, implement a focused Rust solution, validate it under load, and reflect on both the technical and organizational outcomes. Every migration step must earn its place through evidence, not assumption.

# Conclusion

This strategy advisory directly answers the project’s core research question:

**Can Rust-based tools offer a validated, practical performance and reliability advantage over C# for backend CLI and batch processing workloads?**

Based on structured benchmarking, stakeholder input, and migration analysis — the answer is yes.

Rust-based implementations outperformed their C# counterparts in every benchmarked scenario:  
**up to 3.6× faster execution**, **4.7× less memory**, and **3× lower CPU usage**. These performance gains addressed real operational issues including GC-induced memory spikes, unstable parallelism, and high CPU overhead in scheduled batch tools. These issues were also confirmed in consultant feedback, especially around static event leaks and unscalable legacy patterns (con-1).

To validate these outcomes, each tool was tested using repeatable performance profiling (hyperfine), memory and CPU tracking scripts, and functional parity checks across shared input data. Tools were then evaluated using clear migration criteria: performance gain, risk level, ownership, and system fit.

The result is a phased, evidence-based migration strategy.

* CLI tools are ready for immediate replacement
* Batch tools should be phased in with integration testing
* API tools should be deferred until validated through async prototypes

These are not abstract recommendations — they are grounded in tested results, validated pain points, and realistic rollout logic. The outcomes of Sprint 2 establish a clear and actionable foundation for Sprint 3, where the focus will shift to integration, real-world testing, and preparing the first production-ready Rust deployments.