Playing C++ on Nightmare Difficulty

Edouard Alligand – Quasar AI

www.quasar.ai

What we do

The product

- « See into the future »
- A data intelligence platform to
  - capture an unlimited number of events at unlimited speed
  - store them efficiently forever
  - Egress and analysis on demand, regardless of age
- While removing all data engineering headaches!

Use cases and customers

Manufacturing: *Predictive maintenance*

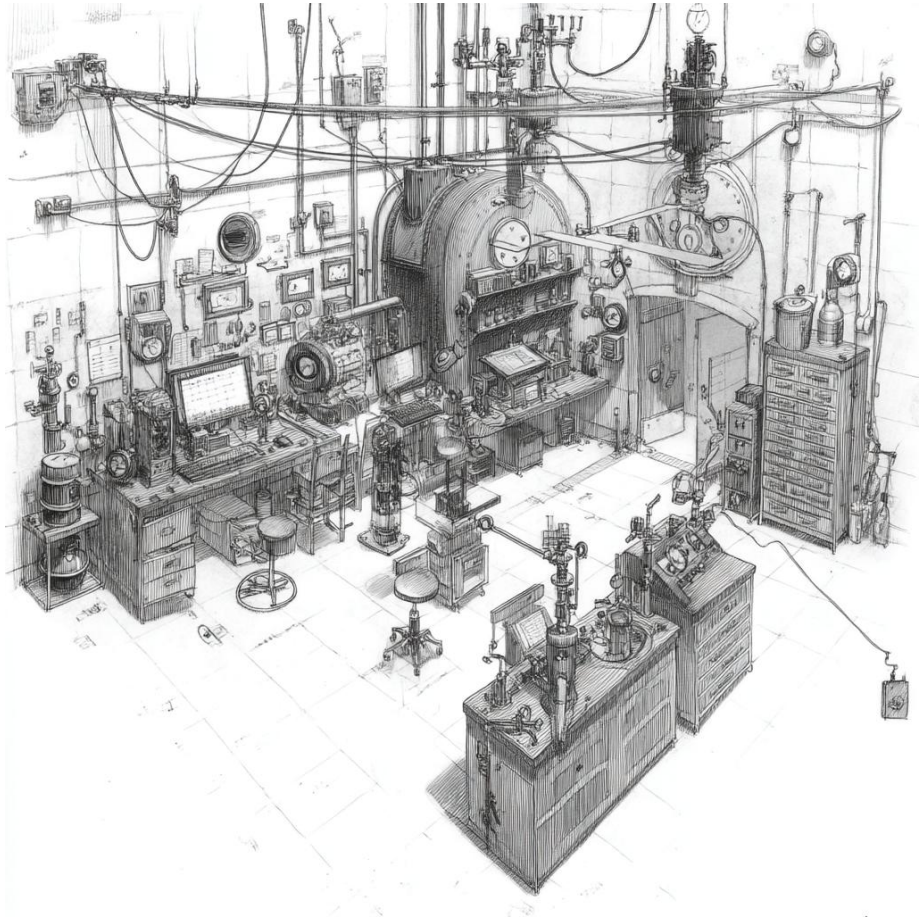
Telecoms: *Cybersecurity*

Finance: *Risk analysis, post-trade analysis, research*

Research: *Simulation capture and replay*

10,000+ deployments

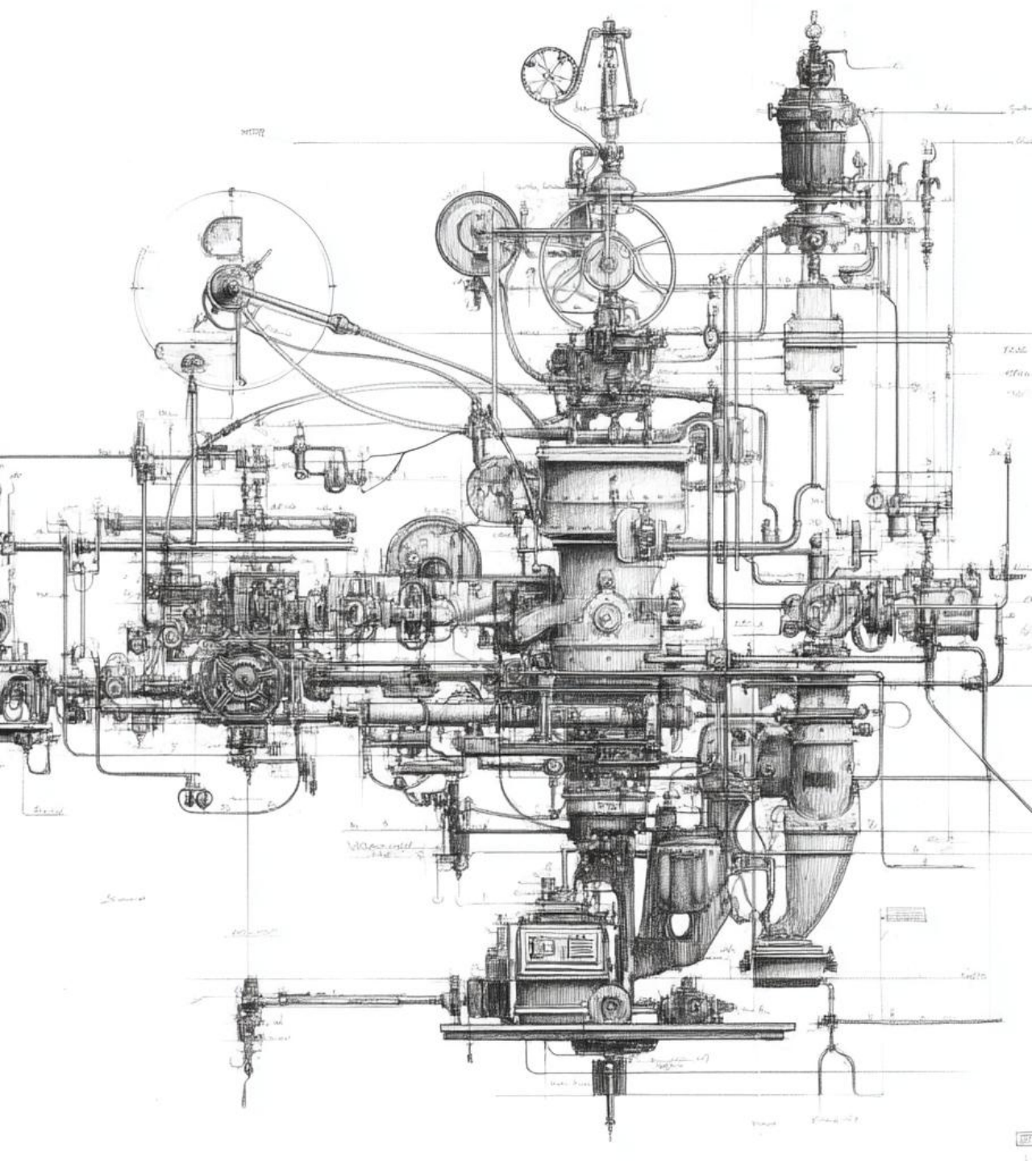
Exabytes of data processed



The Constraints

The features we must deliver

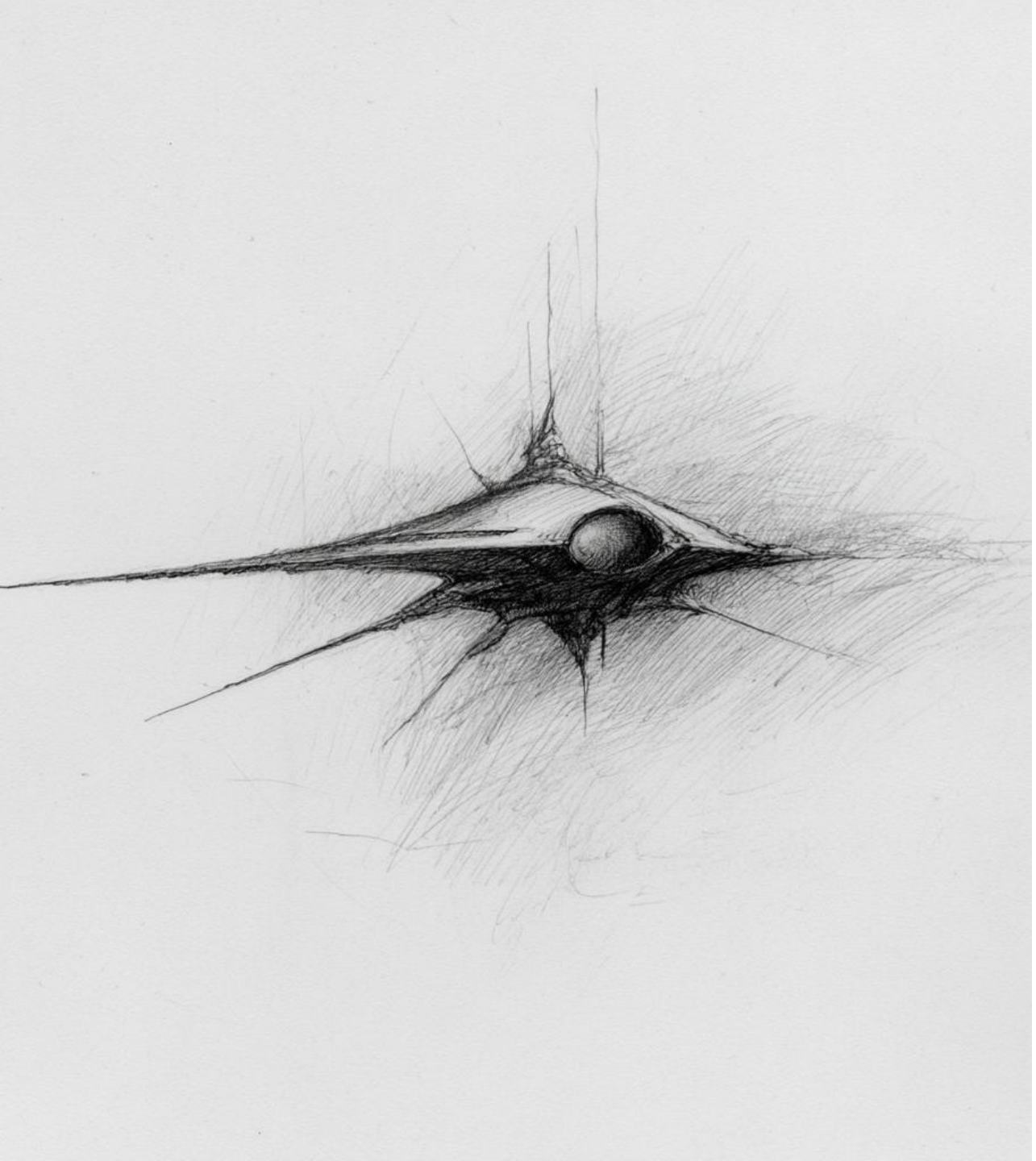
- All you can eat – ingest data as fast as it arrives, egress data as fast as the user can eat it
- Compress data well and quickly to save storage
- Needle in a haystack – access bytes within petabytes instantly
- Run aggregations on large data sets (gigabytes if not terabytes) as fast as possible
- Unlimited capacity – scale out and up
- SQL interface – parse and run SQL queries without the user needing to worry about all the above
- And all of that, ***using as little resources as possible***!



The environment it must run on

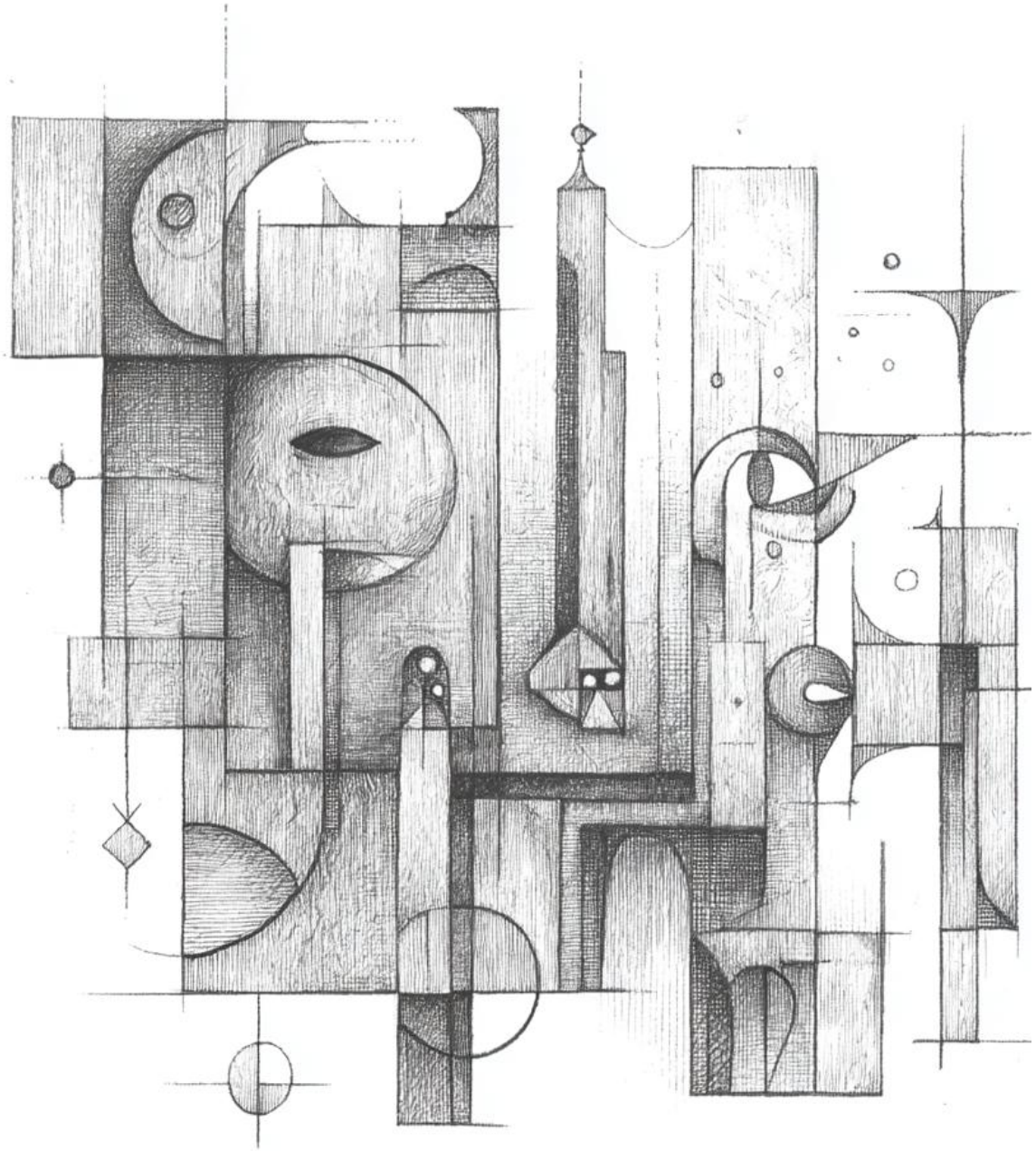
- Must run on 32-bit ARM with moderate RAM and storage devices as well as 64-bit Intel with “unlimited” RAM and storage
- Support NVMe, SSDs, magnetic disks, S3 (Object storage)
- Run on small devices, on-premises servers, and in the cloud
  - Handle the randomness of the cloud
- Linux, Windows, OS X
  - And we added FreeBSD

Why C++?



Objective reasons to use C++

- No restrictions! Complete control. Important for system programming.
- The best performance, with zero-cost abstractions.
- There will be multiple, high performance, high quality, libraries to choose from for any problem you need to solve.
- Extremely mature tool chain, compilers, debuggers, sanitizers, tooling, etc.
- Language is actively improved by a very large community.



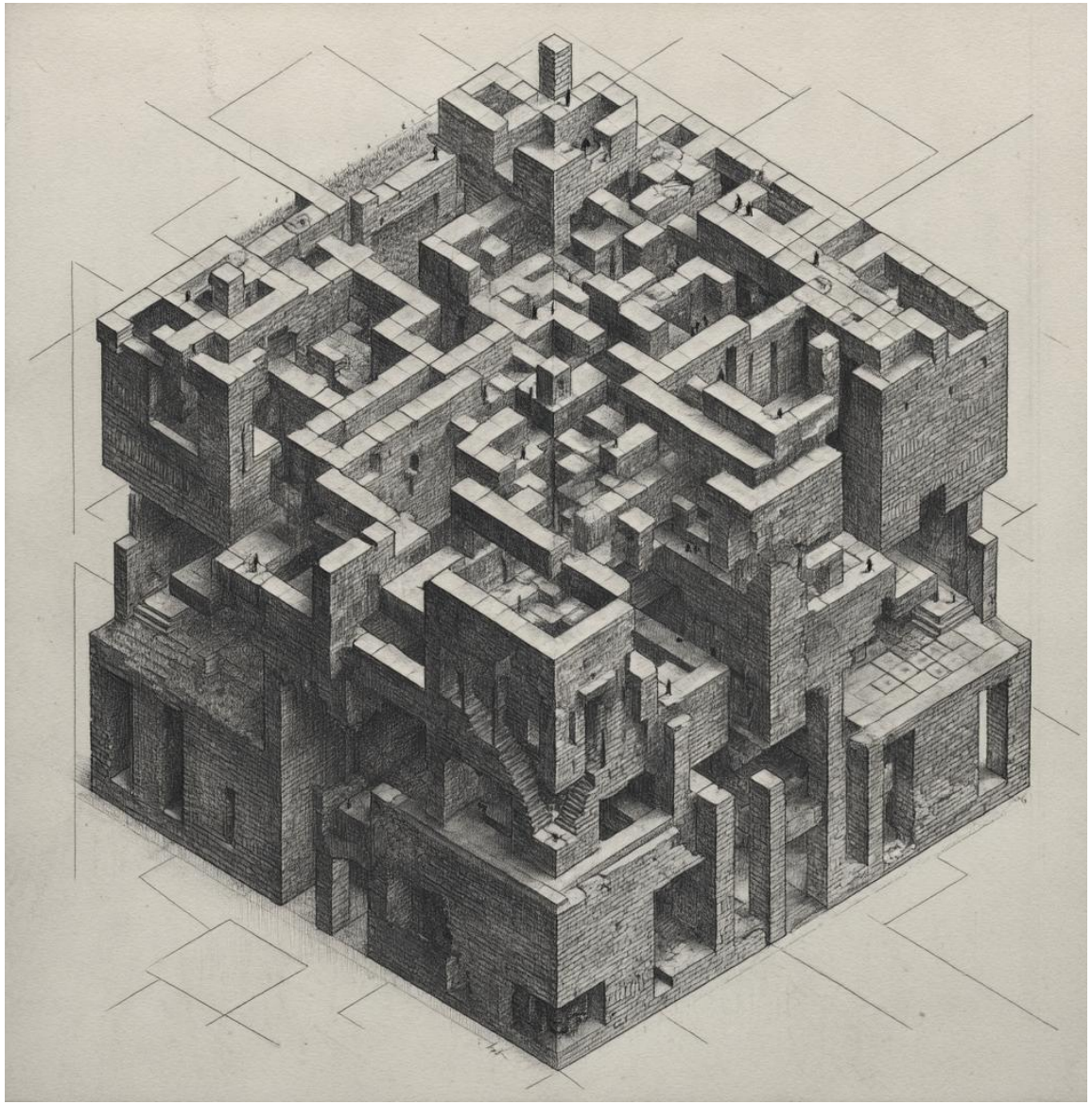
Subjective, but valid reasons

- That's the tool I know the best
- STL is a great library
- Selects for the right kind of developers for the project

But what about <insert language>?

To make things simpler we're going to build a new language from scratch

- Complexity of certain C++ features or behaviors is because of the underlying problem to solve
- Hard problems don't always have simple solutions
- Researching and creating new languages are important but don't fall for the new shining thing



Things that suck

- Package management is way more painful than it should be
- Compilation times can get crazy
- Compilation errors are cryptic and long to parse
  - *Oh, you forgot an include? Let me help you with 100 billion error lines.*
- Being very close to the metal means bugs can be very nasty

Sic transit gloria mundi

001.1 – Assuming thread safety

```
void update_value(int & v)
{
    ++v;
}
```

001.2 – Assuming thread safety

```
mov eax, [v]
inc eax
mov [v], eax
```

001.3 – Assuming thread safety

```
void update_value(std::atomic<int> & v)
{
    static_assert(std::atomic<int>::is_always_lock_free,
"ser i did not order a plate of mutexes for diner");
    // standard even gives you control with std::memory_order
    v.fetch_add(1, std::memory_order_relaxed);
}
```

001.4 – Assuming thread safety

- `std::memory_order_relaxed`
  - I only care about atomicity! No ordering constraints with this operation and other memory operations.
  - Use when
    - Performance matters
    - Ordering does not matter
    - No synchronization between thread is made based on the value on the atomic
  - Don't use when
    - Used as a synchronization primitive (e.g. spinlock or setting a flag after a value has been updated)

001.5 – Assuming thread safety

```
// BAD with relaxed:

std::atomic<bool> ready = false;

int data                = 31337;

ready.store(true, std::memory_order_relaxed); // other thread
may see ready=true before data is set


// CORRECT with release:

std::atomic<bool> ready = false;

int data                = 31337;

ready.store(true, std::memory_order_release);
```

002.1 – SIGBUS

```
// assume p is pointing to a large enough memory area

std::uint32_t decode(const void * p)

{

    return boost::endian::little_to_native(*static_cast<const
std::uint32_t *>(p));

}
```

002.2 – SIGBUS

- Some platforms (e.g. ARM) have very strict memory alignment requirements
- On Intel, these alignment constraints exist for SIMD code

002.3 – SIGBUS

```
std::uint32_t decode(const void * p)
{
    const auto pv = static_cast<const std::uint32_t *>(p);
    std::uint32_t v = 0;
    std::memcpy(&v, p, sizeof(v));
    boost::endian::little_to_native_inplace(v);
    return v;
}
```

003.1 - Random crashes

```
int foo(int & ref)
{
    return ref;
}

int * ptr = nullptr;
foo(*ptr);
```

003.2 - Random crashes

```
int foo(int & ref)
{
    return ref;
}

int foo_ptr(int* p)
{
    foo(*p);
    if (!p)
    {
        // may never run
        std::cout << "p is a null pointer" << std::endl;
    }
}

int * p = nullptr;

foo_ptr(p);
```

003.3 – Random crashes

- The first one is easy because the pointer and the reference are next to each other
  - In real life the problem can be harder to identify
- The second one? The compiler may remove the null pointer check because of UB
- Solution?
  - Sanitize pointers before doing anything
  - GCC : -fno-delete-null-pointer-checks

004.1 - Lifetime issue ASIO

```
// assume a shared_ptr "client"
auto retry = [client = client](error_code ec)
{
        // work on client
        if (ec)
        {
            client->close();
        }
}

// assume an async call with ASIO
boost::asio::async_call(client->socket(), [client = client](error_code ec)
{
        // work on client
        if (ec)
        {
            client->close();
            client = make_client();
            // try again with the new client
            boost::asio::async_call(client->socket(), retry);
        }
);
```

004.2 – Lifetime issue with Asio

```
// WARNING the shared pointer to client is captured here
auto retry = [client = client](error_code ec)
{
        // work on client
        if (ec)
        {
            client->close();
        }
}

// assume an async call with ASIO
boost::asio::async_call(client->socket(), [client = client](error_code ec)
{
        // work on client
        if (ec)
        {
            client->close();
            client = make_client();
            // try again with the new client
            boost::asio::async_call(client->socket(), retry);
        }
);
```

004.3 – Lifetime issue with ASIO

- Asynchronous I/O unlocks a whole new level of performance.
- Shared pointers are a great way to solve lifetime issues in asynchronous code, just make sure you work on the right instance.
- C++ is pain, the sooner you accept it, the easier it will be.
- Asynchronous code is asynchronous pain, the pain just comes later.
- No easy way: careful development, testing, code review, thread sanitizer, etc.

004.4 – Lifetime issue with ASIO

```
boost::asio::async_call(client->socket(), [client =
client](error_code ec)
{
    if (ec)
    {
        client->close();
        client = make_client();

        auto retry =
            [client = client](error_code ec) {
                if (ec)
                {
                    client->close();
                }
            }

        // retry will use the correct client
        boost::asio::async_call(client->socket(), retry);
    }
);
```

005.1 - Where's my SIMD?

C++

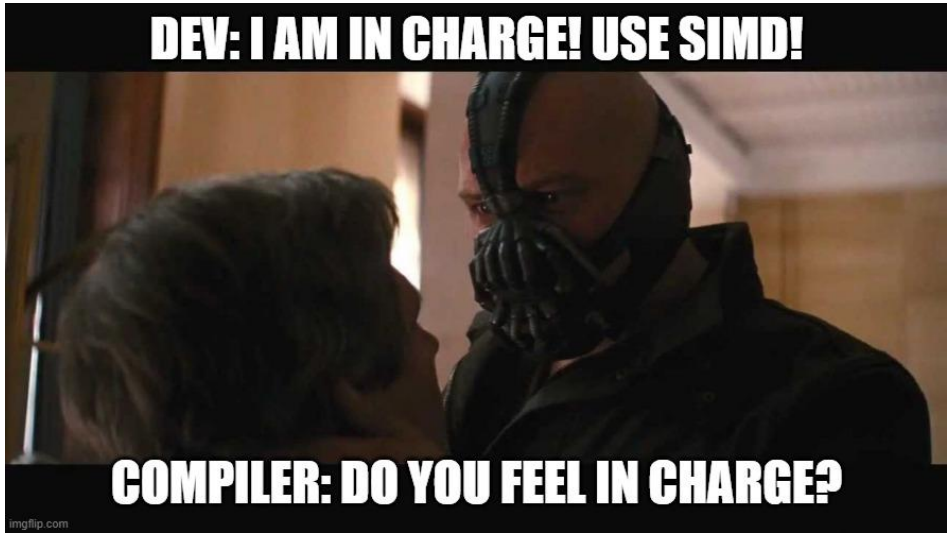
```
float get_first_element(__m128 v)
{
    return _mm_cvtss_f32(v);
}
```

Assembly

```
_Z17get_first_elementDv4_f:
.LFB6474:
.cfi_startproc
    endbr64
    ret
.cfi_endproc
```

005.1 – Where's my SIMD?

- The compiler isn't obligated to honor your intrinsics
- The optimizer can decide than non-SIMD code is more efficient
- Why do you care?
  - You might be led to believe your SIMD code is better than it is (or useful)
  - Might get in the way of optimizing your SIMD code
- Solution?
  - Put code in assembly file if needed



007.1 - Standard ambiguity

```
//  get the epoch

auto epoch =
std::chrono::system_clock::now().time_since_epoch();

// send it to a peer

send_network(epoch);
```

007.2 – Standard ambiguity

- Until C++ 20
  - The epoch of system_clock is unspecified, but most implementations use Unix Time (i.e., time since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970, not counting leap seconds).
- Since C++ 20
  - system_clock measures Unix Time (i.e., time since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970, not counting leap seconds).

007.3 – Standard ambiguity

- In this case use a C++ 20 conformant library
- When you are faced with surprising cross-platform behavior, it can save you a lot of time to refer back to the standard (every word matters)

008.1 – Atomic performance issue

```
struct alignas(64) my_struct
{
    std::atomic<int> one;
    std::atomic<int> two;
};
```

008.2 – Atomic performance issue

- False sharing, one and two are potentially sharing the same cache line, updating a counter “locks” the other.
- alignas(64) at the class level specifies that the object must be aligned on a 64 bytes boundary.
- In C++ 20 you can use alignas() on fields, no need for manual padding.

008.3 – Atomic performance issue

```
struct my_struct
{
alignas(std::hardware_destructive_interference
_size)
      std::atomic<int> one;

alignas(std::hardware_destructive_interference
_size)
      std::atomic<int> two;
};
```

009.1 – A MT bottleneck

```
std::shared_ptr<S> get_object()
{
    static std::shared_ptr<S> instance;
    return instance;
}
```

009.2 – A MT bottleneck

- Code looks like a classic factory generator and can be ignored in a review
- The code is thread-safe, but the control block isn't lock-free, leading to contention
- Additionally, the shared pointer here offers unclear benefits and reference count can create another bottleneck

009.3 – A MT bottleneck

```
std::once_flag flag;
std::unique_ptr<S> instance;

S & get_object()
{
    std::call_once(flag, [] {
        instance = std::make_unique<S>();
    });
    assert(instance);
    return *instance;
}
```

010.1 – Inconsistent performance

```
std::vector<int> v;

for (/*  something*/)
{
    // stuff
    v.push_back(/* v */);
    // more stuff
}
```

010.2 – Inconsistent performance

- The growth strategy of `std::vector` is platform specific
- Generally speaking, reserving your vector in advance is best
- If you have no way to know in advance and notice allocation pressure, consider `std::deque`

011.1 – Inconsistent results

```
std::unordered_map<std::string, int> m;

for (const auto & [k, v] : m)
{
    std::cout << k << ", " << v << std::endl;
}
```

011.2 – Inconsistent results

- `std::unordered_map` gives no ordering guarantee. The hash function plays a huge role in the order, but so does the implementation.
- If you need a consistent order, you have multiple solutions:
  - Same hash function across platform
  - Same hash map implementation (Boost.Unordered)
  - Consider using a sorted container
  - What about a vector?

Per aspera ad astra

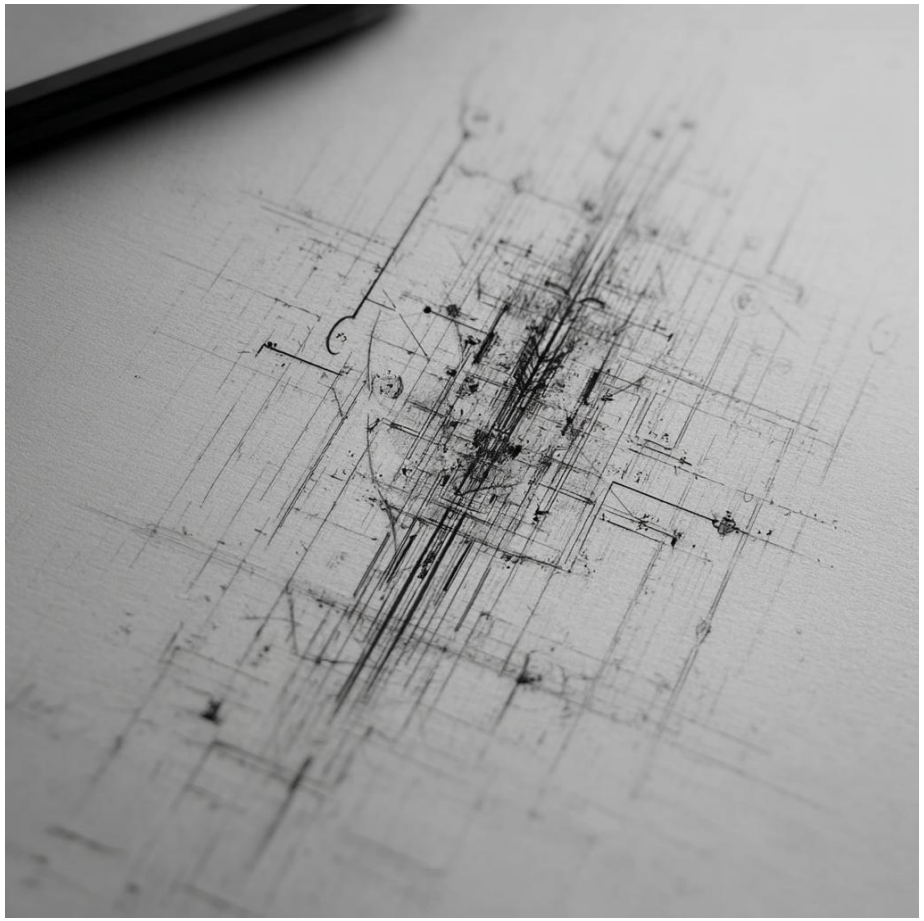
Keep calm and debug the impossible

Sherlock Holmes method

- *How often have I said to you that* ***when you have eliminated the impossible, whatever remains, however improbable, must be the truth****? We know that he did not come through the door, the window, or the chimney. We also know that he could not have been concealed in the room, as there is no concealment possible. When, then, did he come?*

Performance: top down

- What kind of problem am I solving?
- Find the right algorithm
  - Can this be vectorized?
  - Time-space tradeoffs
- The right data structure
  - Start with std::vector
- Perfect the implementation
  - When the profiler says it's worth it



Leverage C++ 20!

- Static assertions
- Const everything
- Annotations ([[nodiscard]])
- Concepts
- Meta-programing
  - Compile-time verified serialization
  - Compile-time verified message dispatch table
- Avoid byzantine C++ (e.g. paradigm overload)
  - Especially with error management
- Avoid OOP patterns

Leverage your STL!

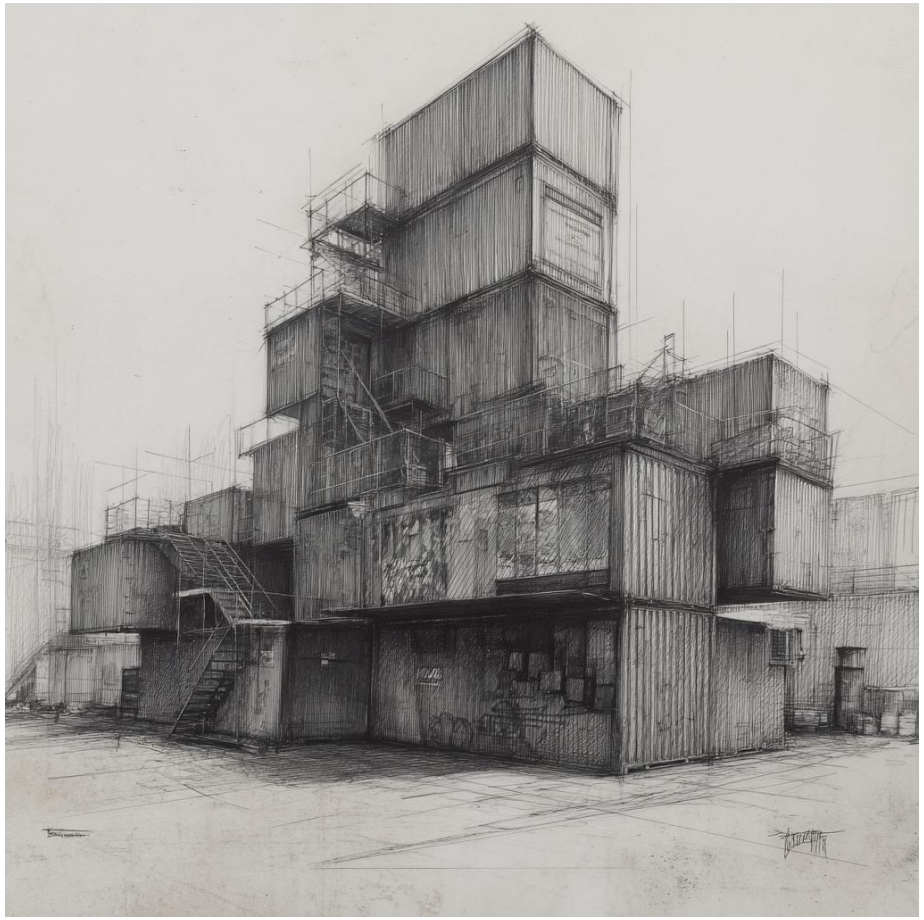
- MSFT
  - Debug
    - `_SECURE_SCL=1`
    - `_HAS_ITERATOR_DEBUGGING=1`
- libc++
  - Debug
    - `_LIBCPP_HARDENING_MODE=_LIBCPP_HARDENING_MODE_DEBUG`
    - `_LIBCPP_ENABLE_THREAD_SAFETY_ANNOTATIONS`
  - Debug & Release
    - `_LIBCPP_ENABLE_NODISCARD`
- libstd++
  - Debug
    - `_GLIBCXX_DEBUG`
    - `_GLIBCXX_ASSERTIONS` (5-6% perf impact)
  - Debug & release
    - `_FORTIFY_SOURCE=1` (~0.1% perf impact)

Tools of the trade

- Unixes
  - Clang & gcc sanitizers
  - Valgrind
- Windows
  - MSVC – the best debugger, built-in profiler
  - Boost.Test – “detect_memory_leaks” (only on Windows)

Thirdparty library management

- Use the STL
- If not available, use Boost
- If not available in Boost, decide between
  - Writing our own implementation (if simple)
  - Find a good third-party library
- How we use third party libraries
  - Included in our tree
  - CMake adjusted to fit our source
  - Built with the binaries
  - Static link to avoid conflicts



Memory allocator

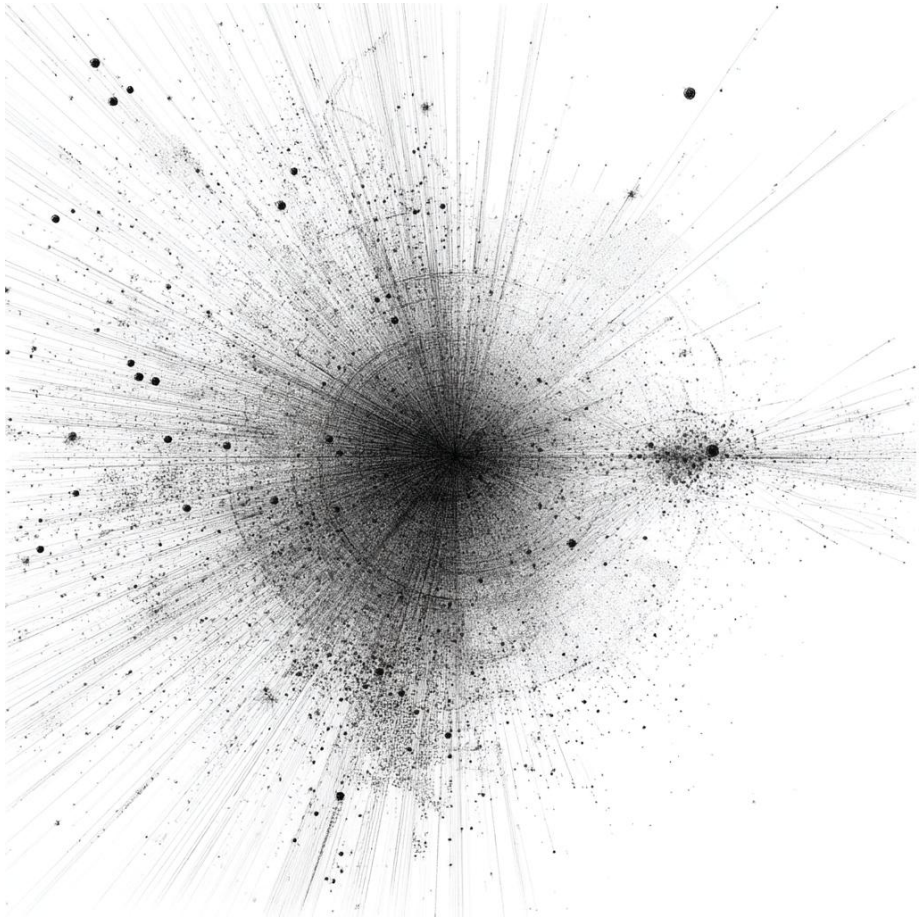
- Minimize pressure on the allocator (reserve, use the stack, pre-allocate)
- Still… The allocator will have a big impact on your performance
- Additionally
  - Consistency across platforms
  - Leaks that are not leaks
  - Memory fragmentation
  - Infamous glibc issue of not returning memory requiring calls to malloc_trim (glibc before 2.22)

Memory allocator

- We use the Intel TBB memory allocator
- Why
  - Same memory behavior across all platforms
  - We have fully reverse engineered the allocator to understand how it works and what to expect
  - Performance for our application is excellent
  - Works well on all our platforms as it's fairly simple
  - Parts of our code is optimized for the allocator (e.g. ensure objects fit in buckets)
  - **No longer use global overloading of operators** as it tends to create issues
    - Instead use STL defined allocators
  - Added some metrics / features
- Also watch out for
  - Jemalloc (Build issues on Windows)
  - Mimalloc (Crash, performance issues last time we checked)

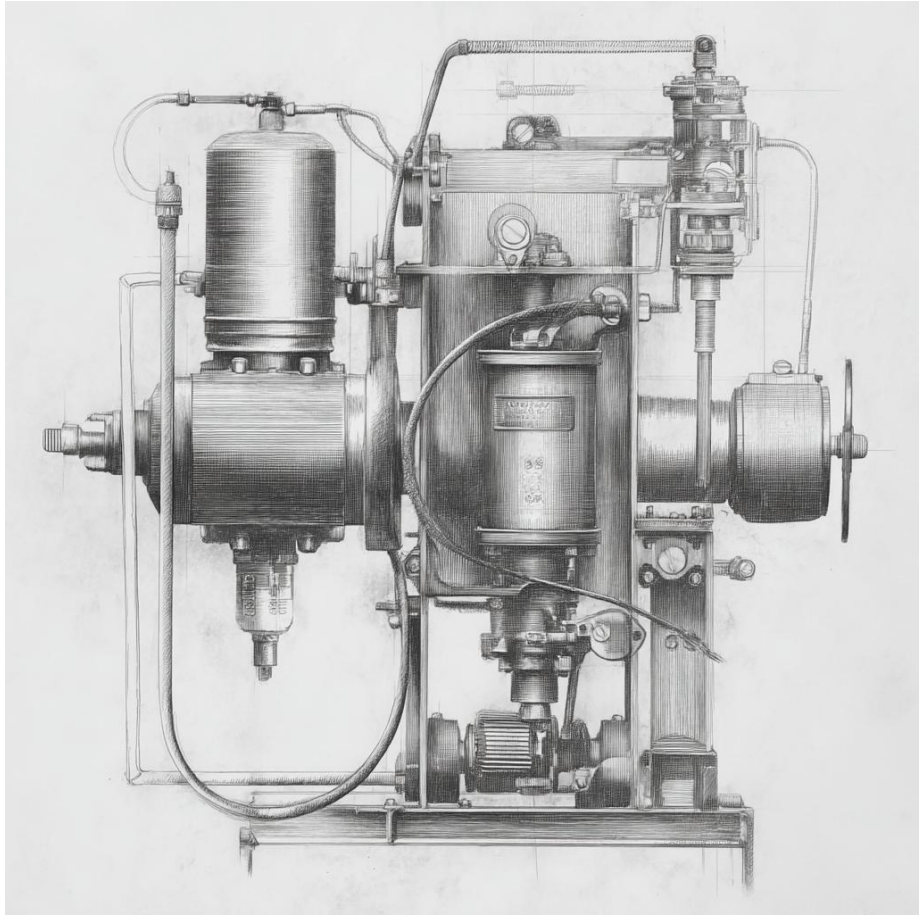
The classics: heisenbugs, race conditions, etc

- Supporting many platforms increases your odds of catching them
- “Loop test”, run the test in a script until it crashes
- We usually follow this process
  - Reproduce
  - Reproduce in debug
  - Reproduce in debug running with the debugger
- Assertions help A LOT.
- Also helps: clang-sanitize, valgrind, etc



Compiler bugs

- Compilation error that shouldn't be here
  - Fix: rewrite the code until it works
- Compiler crash
  - Fix: same as above
- Compile stuff that crashes (sometimes)
  - Last one we had: GCC 13 memmove simd aligned incorrectly, resulting in an error
  - Fix: usually same as above, but takes a very long time



Can’t reproduce, or can’t reproduce in debug

- Don’t give up!
- Document everything
- Not a blocking issue
  - Put it aside until it pops back up
  - You might fix another issue that fixes the underlying problem
- Blocking issue
  - Reproduce as much as the environment as you can
  - Add as much traces as you can that keep the bug alive
  - Rember: memory corruption can happen way before their consequences manifest

Getting the best performance

- Complex systems have a counter-intuitive performance behavior
- Going faster means you're doing less
  - How could I *not* do this?
- Micro-benchmarks and macro-benchmarks are essential
- Make sure you spend time optimizing code that matters

Regressions

- Catch them early – weekly automated benchmarks, continuous integration
- First step: binary search in the change log until the issue disappears
- Don’t mid-curve it: manual QA is essential

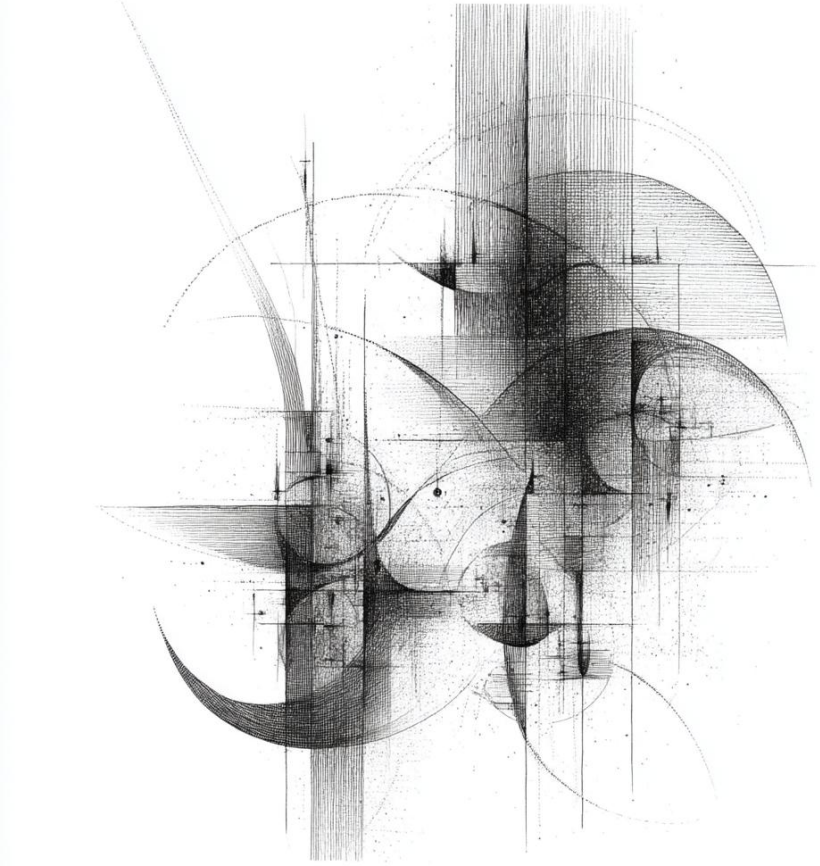
‘All I need is someone to blame!”

Debriefing

- Nameless and rankless debrief
- Fact based
- Root Cause Analysis
- Lessons learned
- Action items

Final word: acceptance

- ***There is no silver bullet***, if you're solving complex problems there's no method, tool, language, framework, or library that will make it easy
- Accept that you will:
  - Make mistakes
  - Have performance regressions
  - Have crashes
  - Make a customer unhappy
- Strive for continuous improvement, be patient



Q&A