Unpopular Opinions
about Software Development

Dino Esposito

CTO Crionet Sports
dino.esposito@crionet.com

NEXT →

How do we build applications that will eventually become legacy systems withstanding the test of time?

Do it right and avoid technical debt

NEXT →

What Does It Mean “Right”?

- **Abstraction and clean Separation of Concerns**
- **Test-driven development**
- **Design patterns scattered throughout**
- (over) **Engineering for future compatibility**
- **Packed with structure and features that might be useful**
- **Event-driven architecture and microservices**

While trying to do it right, are you sure you're not actually adding technical debt instead?

(albeit with the best intentions)

NEXT →

REPHRASE: What Does It Mean “Right”?

- **As simple as possible, but no simpler.**
- **Enough code and architecture to give substance to user stories**
- **Don’t plan (in advance) for any big features**
  - Aim at making customers’ life better
  - Improve and/or streamline processes
- **Single-threaded stories**
  - Transactional, end-to-end paths
  - Limited conditionals (or no conditionals at all) if ever possible

Conditions to Do It “Right”

- **As a product owner**
  - Deep knowledge of the business domain
  - Ability to reduce the problem to clear and explicit terms
  - Ability to negotiate solutions within the boundaries of solving the problem
  - Have engineers fully understand the boundaries of the problem

- **As an engineer**
  - Software modeling skills
  - Ability to enlarge the existing model to incorporate new features
  - Domain skills to challenge new features with well-founded arguments

---

The key is to transform **customer-specific** use cases into configurable **product features** that can be toggled on or off for each installation.

Side Effects of Doing It “Right”

- **Acceptance tests are not specs**
  - Only a checklist to validate whether the problem defined was actually solved
- **Engineers—not product owner—to spot edge cases**
  - It’s an issue if the product owner is the only one able to do that
- **Synced knowledge between product and engineering**
  - What PO think engineers know is not what engineers may know
  - What PO think engineers need to know may not be what engineers need to know
  - Gaps discovered only when engineers deliver (and is not what PO thought)

Having everything nailed down from the start is… like waterfall.

(and is not realistic anyway)

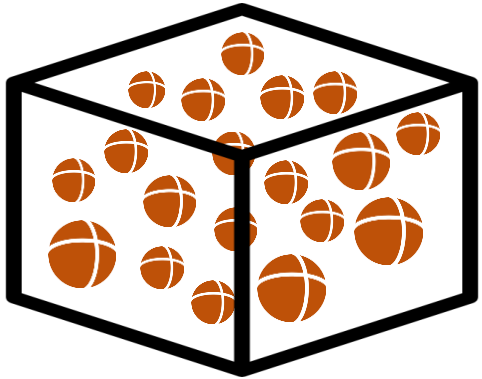
NEXT →

ARCHITECTURE

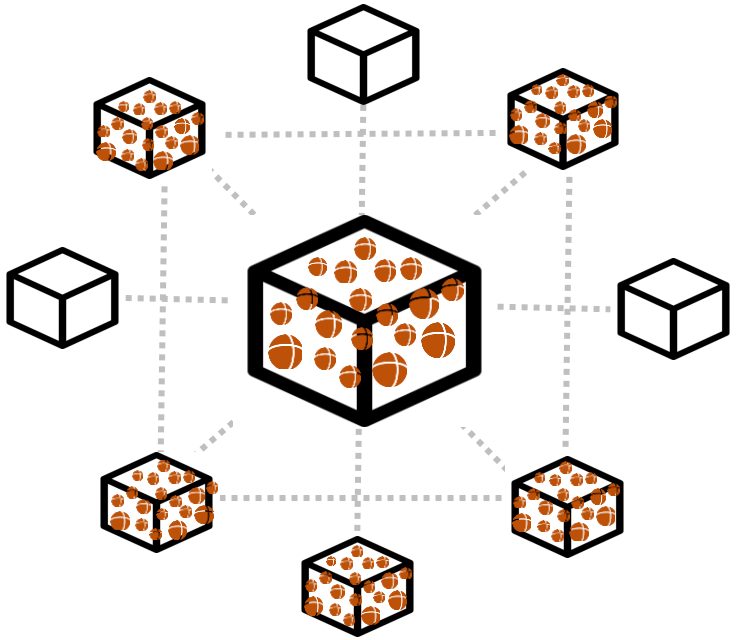
NEXT →

Cloud-native Apps **vs** Enterprise Ecosystem

The scale of the Microservices pattern



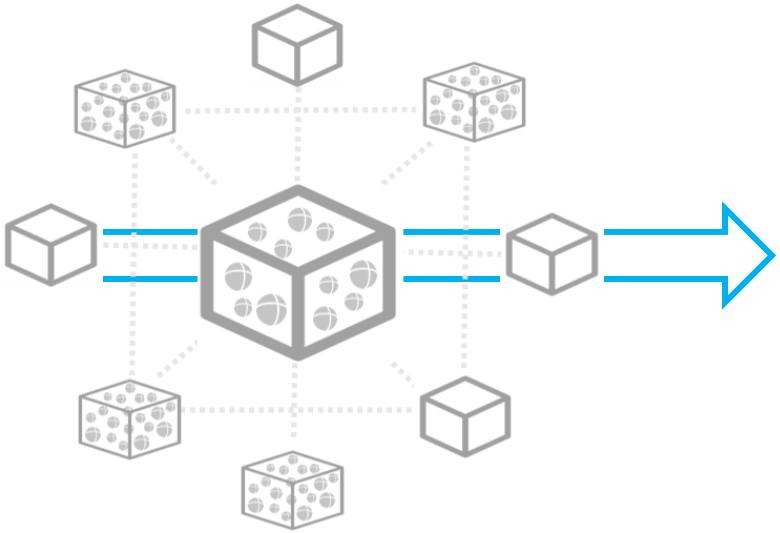
Single LoB application

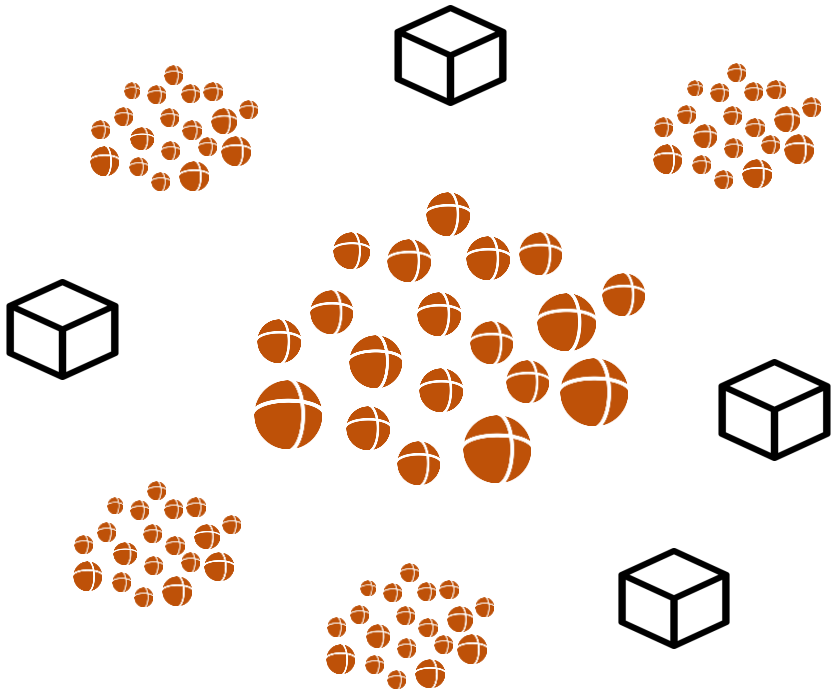


Tens of connected LoB applications

Wonderful Explosion

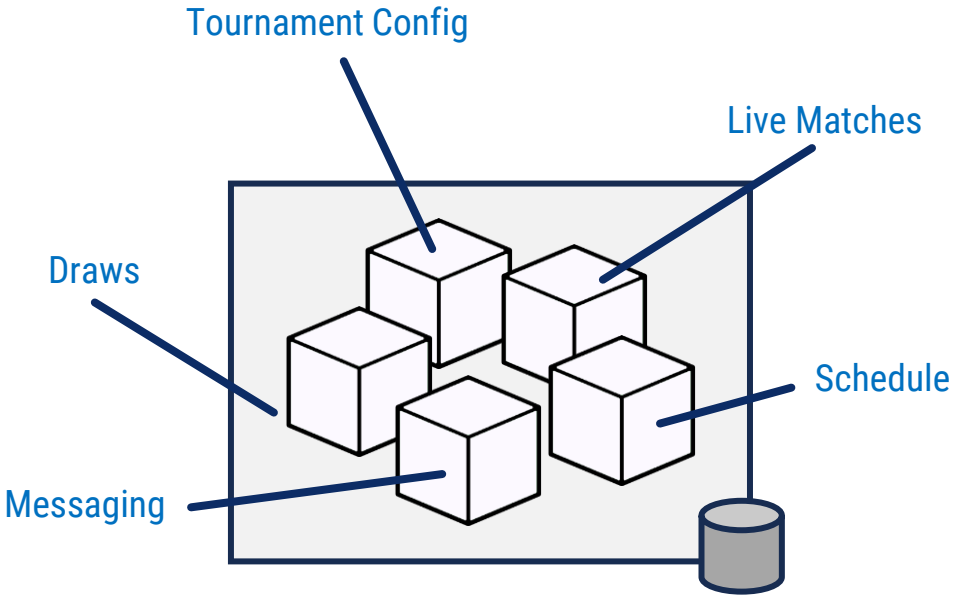
of complexity

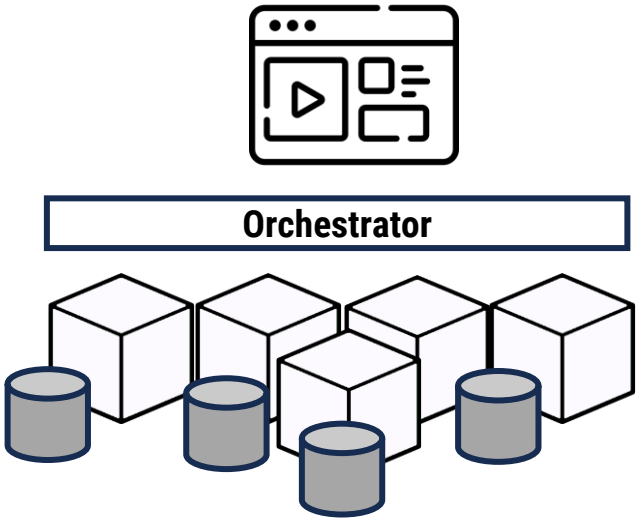




The impact of the Microservices pattern

Tournament Operations Example

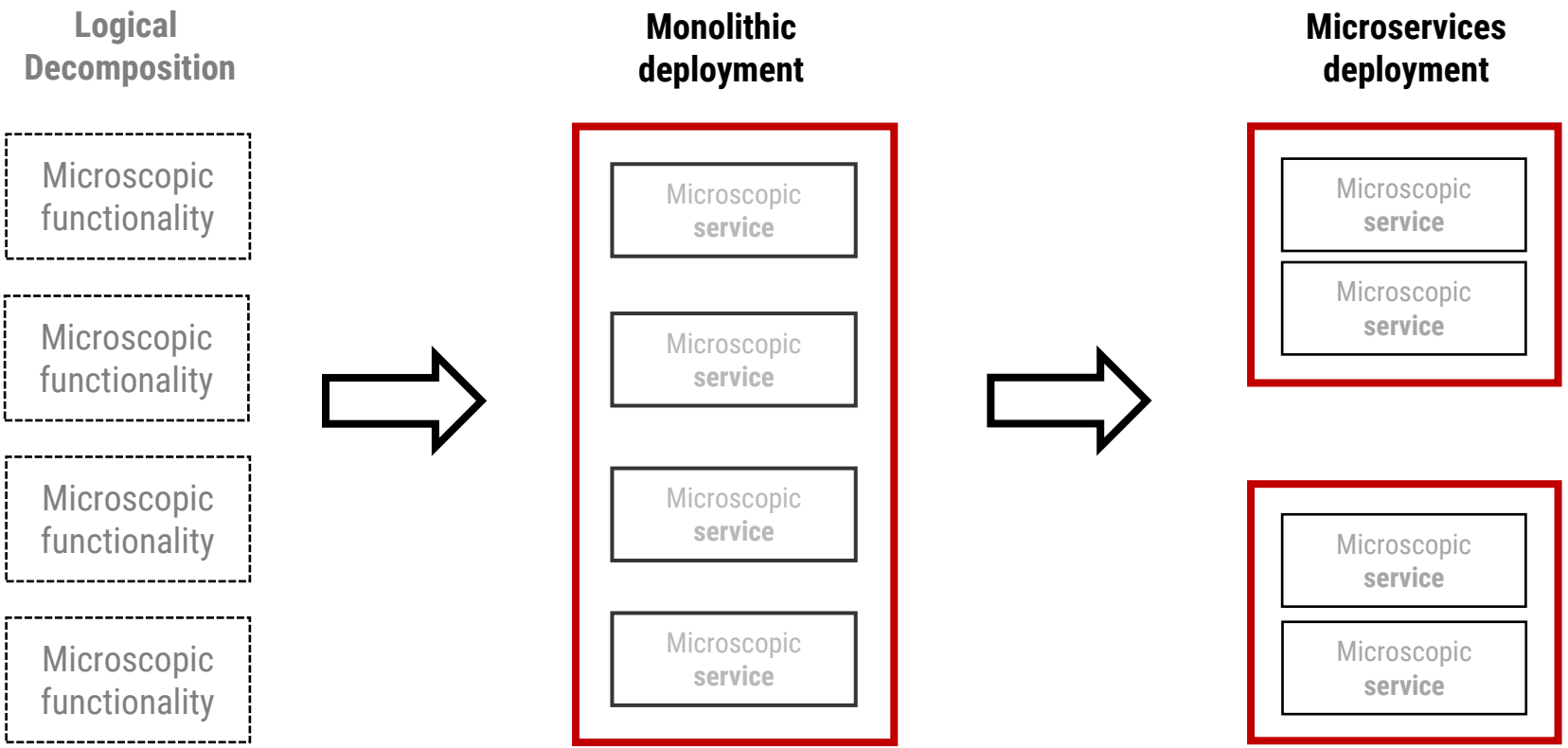




NEGLECTED ISSUES

Latency, additional cloud costs, data collection, logging

Even the most distributed app was a monolith



Even the most distributed of today’s apps started as monoliths

MONOLITHS

- **If a monolith works, go for that**
- **Don’t even think of doing CRUD-ops with microservices**
- **Modular monoliths**
  - Clean or feature-driven architecture done well
- **Microservices** (lambda-size) **require a different app architecture**
  - Cloud-nativity
  - Ad-hoc cloud solutions or Nuvolaris.io for a cross-vendor cloud-nativity

TESTING

NEXT

Testable Code

- **TDD is more about development than catching bugs**
  - Unit tests rail toward the next step of coding
- **Tests are a byproduct of TDD**
  - Written to drive implementation of behavior
  - Not because developers are acting as QA
- **TDD mindset counts much more than actual practice**
  - To be able to write tests you must have testable code
  - Testable code has isolated dependencies and only input data to do the job

On the Role of Unit Tests

- **An engineer/developer is not a tester**
  - Usually focused on logical solutions
  - Struggles to think in unconventional or adversarial ways
- **A tester has a different mindset**
  - Attempts to break the code as if performing a penetration test
- **Unit tests are guarantee of “nothing”**
  - If relevant to what they verify, sufficiently comprehensive in coverage, correctly written, maintained, and kept in sync with the code, tests **just ensure** that the test cases pass
  - Unit tests only find the bugs you thought of

We don’t do unit tests, if not occasionally.

NEXT →

How to Survive Lack of Unit Tests

- **The entire team know the business domain very well**
  - Business onboarding is a must
- **We know why we write the code we write**
  - Developers onboarding is merely a matter of mastering the business domain
- **Yes, we accept the risk of regression**
  - We're up-and-running nearly 24x7, 50 weeks a year
  - We're aware it may fail and we're ready to react to regression
  - We (still) try to write and test code ourselves (like testers, for what we can)

We behave like actors on stage preparing for the show

- We study our script and play it on test machines…
- We go through a series of rehearsal sessions
- How many? Until we all feel confident it would work in the real world.
- We fix bugs quickly and in very few weeks of production we’re OK

Let’s Talk Regression Handling

- **Automated tests and CI**
  - Individual units of code work as expected
  - Catch regressions at the smallest level
- **Pull requests**
  - To review and discuss code changes before merging them into the main branch
- **Branches**
  - To merge at some point

Ways to survive regression

- **Smoke testing**
  - Quick set of tests to ensure most critical functionalities work after any change
- **Incremental development**
  - Small & frequent changes
  - Feature flags (on/off)
- **Prompt responses**

No one cares if your features work in staging. But everyone cares if they work in production.

Once you deploy, you aren't testing code anymore: you're testing systems.

Systems made up of users, code, environment, infrastructure at a given point in time.

NEXT →

“I test in production”

- **Test and Staging environments are just an illusion**
  - Cannot replicate the complexities and surprises of the real world, so the real stress test only comes when the system is in production
- **Pre-launch perfection is a mirage**
  - Thinking you can avoid every problem by thoroughly testing before release is just wishful thinking as every deployment is, in fact, an experiment
- **Testing in production is essential**
  - A strategy for building resilient and reliable services
  - Need to be ready to handle them with speed and flexibility

Any bug is not fixed
until it’s fixed in production

NEXT →

One proven way of doing things is more than enough

https://github.com/youbiquitous/project-renoir/