IfThen

The Hidden Costs of Polyglot Microservice Implementations

By Rob Scott,
Rjae Easton,
and Keith Frey

This is the initial installment in an occasional series of papers from IfThen on topics related to distributed systems and microservices development. Future papers will address topics including testing strategies and service autonomy.

One of the extolled virtues of a microservice architecture is that it allows services to be implemented independently using different technology stacks. This approach, known as polyglot implementation, is possible because microservices interact through well-defined interfaces (typically either RESTful APIs or asynchronous message passing) meaning the developers of each service are free to implement the internals of the system in whatever language or technology stack they choose.

Polyglot implementations do offer several advantages. They foster team and service autonomy, provide organizational flexibility, and enable teams to choose the best tools for the job at hand. While we agree there are there are numerous reasons to favor polyglot implementations, this approach also carries with it both direct and opportunity costs that should be evaluated within the overall context of the project including product needs, budget, and release schedule.

Rebuilding infrastructure for each tech stack

When teams choose to use different tech stacks across their microservice implementations, they often have to rewrite a substantial amount of infrastructure code for each stack. This is especially true if not all stacks implement a particular standard that the organization is adopting (for example, OpenTracing).

Infrastructure code that might need to be re-written includes foundational libraries the team uses for validation, shared classes, wrapping of external dependencies, resilience management, base classes for services, etc. For non-trivial services, this effort can be significant.

Infrastructure costs are not limited to the run-time environment and implementation. Significant costs can also be incurred by the overhead of managing the build and deployment scripts for various stacks. Each stack typically has different commands to execute, and mundane details like container sizes will differ across stacks and languages.

This infrastructure work has both direct costs—tied to developing and maintaining code and scripts for different stacks—and opportunity costs tied to the effort spent doing work not directly related to implementing business functionality.

Cross-stack debugging is hard

When operational issues require debugging across microservices and stacks, it's unlikely one person can do the debugging. A C# developer may not be able to debug a service developed in Python by another developer and neither of them will know what's going on with Scala. Even for languages that are very similar like C# and Java where developers can generally read each other's code, the development environment, configuration, libraries and idioms can differ wildly. As another example, if one team used Entity Framework and another used Hibernate, finding someone who understands both at a deep enough level (for any but the most trivial issues) is quite difficult. The overhead of cross-stack debugging is a direct cost akin to a tax that is levied on your development team.

Inconsistency leads to operational issues

Different technology stacks handle errors, perform logging, and provide instrumentation in ways that vary across implementations. In order to present a unified view for operations, significant work may be required to merge the views or to create libraries that standardize the information provided. Two costs have to be traded off here. The first is the extra effort that is incurred by the operations and development teams to understand and isolate issues—this cost is a hidden tax levied on each issue that has to be investigated. The second is the direct cost of writing the code to standardize the views. You'll pay (at least) one of the two in a polyglot implementation.

Systems Management costs are increased

Multiple technology stacks can increase the overhead of managing system level services such as databases (remembering that each microservice has its own independent data store). For example, if one service is using SQL Server, another is using Postgres, and another RavenDB, you need to have the ability to manage all of the aspects of availability, sharding, backups, etc. for each of those systems.

Conclusion

We're not arguing that you shouldn't use the right tool for the right job and that different stacks aren't appropriate for different jobs, rather we're emphasizing that:

— You need to understand the hidden costs of polyglot implementations.

  There are valid reasons for using polyglot implementations such as experimenting with a new technology to meet business needs or because you have different types of work that you need to do (more on that below), but there are associated costs with making those choices.

- You need to classify the jobs that you need to do and select your tools and tech stacks appropriately.

  Tools tend to excel at certain types of jobs. For example, if your service needs to do a job that involves a lot of scripting, you might prefer to use Python. If you're doing work that benefits more from static typing, your stack of choice might be Java or C#/.NET. While you may need to use different stacks for different types of jobs, you should try to use the same stack for similar types of jobs. This type of approach will lead you to less duplicative work building out infrastructure for the stacks since they will tend to be doing much different work.

- You shouldn't be making tech stack choices solely on the current preferences of the team or based on their current skill set. You have to factor in the other costs which tend to overwhelm the cost of language choice over time—every new stack imposes another set of switching costs and technical debt that can inhibit your ability to adapt in the future.

All development projects exist in an organizational context that includes time to market aspirations and a budget in addition to product needs and technology factors. The inevitable tradeoffs among these factors means that you need to optimize the way you spend your money, time, and talent. Adopting a polyglot implementation strategy may be part of an optimal solution for your project, but we recommend keeping your eyes open to the sometimes hidden costs.

Bios

Rob Scott has been building and studying distributed systems for more years than he cares to admit. He has built products and systems in industries as diverse as telecom, workflow management, finance, healthcare, transportation, and insurance. He's particularly interested in the application of domain-driven design and other requirements analysis techniques to building microservice-based systems.

Rjae Easton was drawn to the craft of software engineering over thirty years ago and has created countless systems since then. These include applications for financial markets, life sciences, consumer products, real estate, and more. Rjae enjoys the challenges of delivering a successful project, and it still amuses him how simple it is when a few key factors are done right. This is probably the aspect of software engineering he enjoys most: helping stakeholders succeed.

Keith Frey has over twenty-five years of leadership experience in software development, R&D, and program management. Today Keith leads engagements for IfThen including distributed systems and microservices development for medical payments and B2B auction applications. In the past, Keith was a VP at Turner Broadcasting System where built innovative consumer-facing products and technology platforms for large Internet and television audiences.

Keith Frey
Senior Vice President

kfrey@ifthen.com
404.623.8331