Table of Contents

John Lewis IT Software Engineering Principles

Through our combined experience of building and releasing software we have discovered and come to value the following principles. These principles are not hard and fast rules but rather some things that we apply and use to guide us on a daily basis. These principles could also be used in a "discovery phase" when selecting a new product to purchase.

We created these principles to:

- Guide our existing engineers and help them to make decisions aligned with our thinking
- Introduce new engineers to how we do things, enabling them to make decisions aligned with our thinking
- Share with colleagues beyond our Technical Profession, to help them see why we make the decisions that we do

Many of our Engineering Principles are inspired by the research and recommendations found in the book Accelerate – The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations.

We have borrowed the style of principle used by TOGAF's Architectural Principles.

Printable version

A PDF version is available.

Invitation to contribute

If you're involved in the software engineering process at John Lewis, then we want your input! The principles are maintained in a GitLab project tech-council/engineering-principles.

For correction of typos, fixing broken links, or anything similarly unlikely to raise any debate, please fork the project, make the change and raise a merge request.

For more significant adjustments to an existing principle, do the same, but perhaps anticipate some discussion on the merge request.

For suggestions of new principles, or fundamental challenges to existing ones, please start a conversation in #comm-victoria-engineering-group

Self-assessment

Here are some questions you might find useful when running a self-assessment with your team against these principles.

Build Differentiators

- Do you consider this when choosing the technology?
- Google Cloud Datastore or Elasticsearch are examples of "buy"
- Do you reinvent the wheel, building systems already out there?

Design for Emergent Reuse

- If there are lots of different components do you take into account the pace of change of the components?
- Do you over-engineer your solutions?
- Do you reuse for the sake of it?
- Is your system going to be around for a long time?

Evolutionary Systems

- If you change your system is it an isolated change with regard to other systems?
- How many people does it take to release?
- How often can you release?
- Can Ops release during the day?
- How many requirements do you put into one iteration?
- How future proof is it for requirements that may not exist?

Scale Horizontally

- State cannot be stored in-memory but rather must be persisted to the client or a shared datastore.
- It is not always possible to scale horizontally e.g. for traditional relational databases, increased performance is often achieved through better hardware or higher specifications
- Services should strive to be idempotent

Small and Simple

- How long do your tests take?
- Can you describe your components?
- Can you test your components independently?
- Can you deploy components independently?

Smarts in the Nodes, not the Network

- Do you have business logic in your integration layer?
- Do you apply business rules in your data transformations?
- If there is business logic is it for your own system’s consumption?

Cloud Native

- Are you leveraging the services in the cloud?
- Did you intend it to be in the cloud from the start?

Production Ready

- Can you release straight to Production?
- Do you understand your failure modes?
- Do you understand how monitoring helps you understand your failure modes?
- How close is the system to its limit?
- Are there circuit breakers?
- Are there rolling deployments, eg kubernetes?
- Do you have dashboards, golden signals, engineering metrics for traffic and resilience?

Keep Pace with Technological Change

- What technology is the team using?
- Is it up to date?
- Is it well supported?
- Is it current?
- How often is it upgraded and patched?
- Are the team interested in this area?
- Is the software on its way out?

Model the Business Domain

- Does it use DDD (Domain driven design)?
- Could a business person understand the language used in your code?
- How well do you own your domain?
- Are there changes not owned by your team?

Secure by Design

- Have you considered access control?

Consistent Environments

- Are you using infrastructure as code? Examples are terraform or puppet.
- Have you got any processes in keeping data up to date?
- Are there lots of manual processes?

Automate By Default

- Are you using a CI tool?
- What manual gates are in place?

Get Feedback Early and Often

- Are you practicing Continuous Delivery?
- Could you do daily releases?
- Are you using trunk based development with feature flags as necessary?
- Are branches short-lived?

Understandability

- If a new person joins, can they get up to speed quickly?
- Can code be safely refactored due to good test coverage?

Build Differentiators

If functionality is a differentiator for the Partnership then we should prefer to build, rather than buy and/or customise.

Rationale

Customisation of Commercial Off-The-Shelf (COTS) packages, and misuse of Software-as-a-Service (SaaS) beyond its designed-for use cases, can be costly in comparison to building equivalent functionality. It also has a longer-standing impact on the future pace of change, requiring maintenance and development on top of often-complex customisations that require specialist knowledge.

Implications

- Extract what value we can from existing COTS packages and SaaS.
- Use COTS packages and SaaS for their specific strengths,and compose differentiating systems around them as appropriate.
- Legacy systems, where we can differentiate, may need a wrapper layer to facilitate strangulation and subsequent replacement.
- Prepare for today's differentiator becoming tomorrow's commodity. We should watch the market and change our approach when there is value in doing so.

Design for Emergent Reuse

Design for well-defined use cases and adaptability. Address reuse as an optimisation opportunity rather than a goal.

Rationale

This principle is a version of YAGNI applied specifically to reusability rather than functional features. We believe it deserves extra attention in this context as it can conflict with our natural instincts about what good looks like.

Designing for reuse from the outset, in the absence of well-understood use cases, requires us to make assumptions about what will be valuable at some point in the future. These assumptions usually turn out to be wrong. As well as wasting effort and creating a cost of delay, we have seen this approach to reuse significantly inhibit change, by creating unnecessary complexity, dependencies and bottlenecks.

Code that is designed for reuse is generally harder to build, maintain and use than code designed for a single purpose[1], so we should first establish that there is concrete value in reuse outweighing the costs, and accept duplication until then.

Implications

- Identify reuse as it emerges through evolution or modelling the business domain; prefer duplication over premature abstraction until you have evidence new dependencies will not constrain the required pace of change.
- Be sceptical of components that look similar but aren't the same once you consider business use cases and what might trigger them to change and diverge.
- This principle will require particular attention when considering core capabilities with the potential to serve multiple brands or channels.
- This principle can only be effective if reuse can be enabled in the future at a similar cost to today. We must therefore focus on building easy-to-change Evolutionary Systems.
- If reused code becomes a bottleneck, remove the coupling by forking or duplicating the code.
- The risks of reuse are usually lower for generic, domain-independent features such as monitoring, logging, service discovery, configuration, etc. The opportunities and risks of patterns such as Service Templates and Service Chassis should be evaluated in this context, aligned with our approach to technological diversity described in Keep Pace with Technological Change.

Further Reading

> [1]. Building Evolutionary Architectures: Support Constant Change by Neal Ford, Patrick Kua, and Rebecca Parsons ↩

Evolutionary Systems

Systems and architectures should be designed and built to enable easy, incremental change.

Rationale

At this point in time we know less than we'll know in six months. Our software should be able to evolve as we learn more about the business domain, the operational environment, and the technology itself. Change is a constant in the world around us, and its impact is increasingly unpredictable. However, by engineering our software for evolvability as a primary concern, we give ourselves the best chance of experimenting, learning and thriving in new conditions.

Implications

- Whilst we cannot and should not design for every possible eventuality, the evolvability of our systems can be hugely improved, at low cost, if the likely triggers and impacts of change are considered early enough. Uncertainties, "what-if" business & technology changes and failure modes should be made explicit, rehearsed and mitigated where the cost to do so is low relative to the risk. Use Small and Simple and Model the Business Domain to group together functionality in ways that minimise the splash-zone of potential change.
- When making design decisions, give yourself the best chance of adapting to new information by waiting till the last responsible moment, and preferring choices that are easily reversible or that keep your options open.
- Define and regularly measure the architectural qualities that are most important to success, and which must be nurtured as the system evolves.
- Systems and processes will need to be exposed through clearly defined interface contracts so that the underlying implementation can change and evolve without impacting the wider system. (Note: this does not mean that a component or system's *internal* interfaces necessarily need to be approached in the same way.)
- Components must be accessible through open, non-proprietary standards (with a preference for HTTP & in particular REST). Embracing the best practices of the web maximises the opportunity for evolution while minimising the risk of technical redundancy.
- Standards that are derived from working software and not just working groups are preferred. Battle-tested standards used by several organisations tend to be more adaptable than over-engineered hot air. Most successful standards tend to be the ones that are adopted by Open Source communities.
- Adopt patterns such as Tolerant Reader and the Principle of Robustness to enable systems to release changes independently without necessitating parallel change versioning and migration.
- Sometimes gradual/incremental change is insufficient as requirements change, so consider Sacrificial Architecture as a pattern. Don't be afraid to throw away and re-write, as this is often faster and will lead to better quality. If we have kept things Small and Simple this should not turn into a painful Big Rewrite.
- This principle depends on a number of others to be able to apply it in full: Small and Simple, Model the Business Domain, Get Feedback Early & Often, Understandability.

Scale Horizontally

System components should be horizontally scalable where possible.

Rationale

It is cheaper to scale services by adding inexpensive resources like commodity servers or database nodes rather than buying larger and larger pieces of hardware which, at a given point, are unable to scale further. By embracing horizontal scalability early and designing our systems to work in this manner, we eliminate the cost of changing at a later date.

Implications

- Conversational state cannot be stored in-memory of a service instance but rather must be persisted to the client or a shared datastore.
- Provided they meet your needs, prefer “serverless” persistence options from your cloud provider, since these will typically handle the scaling-out of your application with no extra effort. e.g. Google Cloud Datastore rather than Google Cloud SQL.
- It is not always practical to scale horizontally. For example, increased performance for traditional relational databases is often achieved through better hardware or higher specifications.

Small and Simple

Where software is tending towards complexity, look for opportunities to break it down into smaller and simpler systems.

Rationale

Monolithic applications can be successful, but it has become common for organisations to feel frustrations with them as they grow in complexity and scope over time. Change cycles are tied together – a change made to a small part of the application requires the entire monolith to be rebuilt, retested and deployed.

Over time it is often hard to keep a good modular structure, making it challenging to keep changes that ought to affect only one module contained within that module. Often, a bug in one part of the system can affect other parts of the system in unexpected ways. Scaling requires scaling of the entire application rather than parts of it that require greater resources. Scaling requires uplift of the entire application rather than only the parts that require greater resources.

Software which focuses on doing fewer things well is easier to reason about. This benefits its reliability, understandability and makes it easier to test.

Implications

- It is important to be fully aware of the interconnected nature of the overall system before looking to decompose it - premature abstraction can accidentally lead to an overly complex distributed systems if not well understood.
- Build systems around business capabilities that change independently to others, rather than individual features or technologies.
- Aim to decompose your system into a suite of smaller systems, each of which focus on doing a small number of things well.
- Beware the risks of creating a distributed monolith.
- Services are independently deployable and independently testable.
- Reduce the burden of managing several smaller systems through automation.
- In distributed systems, it is important to design for failure.

Smarts in the Nodes not the Network

Sometimes referred to as “smart nodes and dumb pipes”, meaning that systems aim to be as decoupled and cohesive as possible, and not centrally choreographed in middleware.

Rationale

Systems owning their communication with other systems is an important aspect in allowing them to adapt to and keep pace with business change. It allows them to evolve over time without needing to negotiate and coordinate with separate teams and associated delay due to hand-offs, as well as meaning that the knowledge of how a system communicates is contained within the system itself.

Implications

- Prefer open protocols, such as HTTP.
- Implement integration adapters to isolate the technicalities of integration, and anti-corruption layers to isolate the translation of business data, between bounded contexts.
- When data needs to be exposed or exported outside the service boundary, wrap this communication in a service layer rather than allowing consumers to access the data directly. This helps prevent business logic leaking out of the system and avoids accidental coupling that may constrain future change.

Related Reading

- Hexagonal Architecture

Cloud Native

Build systems that are suitable for running in the public cloud, using cloud-native technologies suited to our provider of choice.

Rationale

Use of cloud-native services helps us focus on building differentiators and supporting a rapid pace of change, and reduces operational complexity. This is particularly important where we run a larger set of distributed, smaller systems which need to operate independently.

Implications

- Prefer “Platform-as-a-Service” over “Infrastructure-as-a-Service”. Where possible, use the JL Digital Platform, which has been built with this principle in mind, and provides a curated set of tools and services to achieve this.
- Prefer open tooling for configuration and deployment that works well with our choice of cloud.
- Leverage the services where our cloud provider is strong. Do not overcommit to being cloud-agnostic where this creates a large amount of engineering effort or misses the opportunity to make use of high-quality cloud features.
- Build software to take advantage of cloud features, and be tolerant of cloud failure scenarios. For example, consider variable network conditions and minimising of local state to support automatic scaling.
- Teams should continuously review the cloud services they consume for suitability and cost-effectiveness.
- Compute, storage and network resources are abstracted from applications, isolating them from underlying infrastructure dependencies and therefore improving portability.
- Cloud-Native technologies are focused around APIs rather than low-level infrastructure interfaces.

Data Stewardship

Actively protect and care for the data stored and accessed by a system, at least as much as the functional behaviour of that system.

Rationale

Information is a fundamental asset to the business, critical to its success. It may be re-used or adapted to add significant value in new business contexts, well beyond its original intent, outliving the system that first created it. Inadequate care for data concerns can result in misuse, insecurity and unintentional coupling between systems, generating widespread complexity and flakiness, as well as significant financial and reputational risk to the business. The resulting costs and constraints can be extremely painful and expensive to resolve.

Implications

- Each bounded context must manage the data it uses, acting as a responsible owner, even if it is not necessarily the "master" of that data.
- Teams must understand and address any data privacy concerns related to their data, including policy and legal compliance. For example: classification & retention policies, GDPR, PCI-DSS.
- Where others need to consume an application's data, it should expose it via an interface with an explicit contract.
- Consumers of data exposed by an application should be able to easily find metadata about its meaning and quality e.g. definitions, accuracy, freshness, time-to-live as well as operational service levels e.g. RTO, RPO, support hours. It should be clear who they should contact to ask questions about it.
- Where there is a potential choice of data sources, there is a risk of accidental misuse, complexity and flakiness. Consumers should first model the business domain to identify the bounded context they want information from, then aim to get as close as possible to the commonly agreed source of truth *for that context*. This will usually be the origin or trusted "golden record" that particular business domain relies on operationally.
- Consumers and providers may both need to balance availability and accessibility of the data with other non-functional considerations such as performance, quality, consistency and freshness. These factors are typically in tension and need to be traded off.
- Data can be transformed or translated as it is shared. Applications should represent their copy of data using schemas designed for the context of their business domain and use cases.
- When making significant changes that could be breaking for consumers, including changes in meaning, make use of versioned interfaces.

Related reading

- Data integrity at the origin | Thoughtworks Technology Radar
- APIs as a product | Thoughtworks Technology Radar
- How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh
- Data considerations for microservices | Azure Architecture Center

Production Ready

Systems are engineering for use in Production from the start – they are scalable, observable and tolerant of failure.

Rationale

Building more, smaller systems and utilising cloud environments means understanding and adapting to change, and handling failure, is more important than ever. Ideally, a system is self-healing or produces only relevant, actionable alerts.

Implications

- The system should have sufficient monitoring, logging and alerting to be able to understand and report on its health, and the health of its dependencies, in a consistent way. Anyone in JLP should be able to access these telemetry tools.
- It is important that alerting and diagnostic tools are targeted and relevant in the information that they provide. Overloading dashboards with information impairs usability, and excessive alerting leads to fatigue and important notifications being missed. Analytics data is often best surfaced through separate tools from those used for operational purposes.
- Systems should gracefully degrade on disruption – for example, by using circuit breakers, bulkheads, caching or partial responses. Likewise designing calls for idempotency can help avoid issues with transient failures or unforeseen behaviours.
- Teams and business owners will need to work together to identify relevant business and technical KPIs and SLOs for their product.
- Teams should understand and improve the resilience of their service through failure mode & effects analysis, and by experimenting with intentional faults.
- Dashboards need to be implemented for the most important SLOs.
- Not all risks to Production readiness can be analysed in advance so exploratory testing, Production observability, fault injection, post-incident reviews, and operational monitoring should be used to expose new information about software behaviour.
- Teams should aim to release new services into Production as soon as possible, and software should be kept deployable throughout its lifecycle. As well as the benefits arising from faster feedback through Continuous Delivery, this helps expose teams to the release processes and tools in use in Production earlier, so that they are well-understood in advance of launch.

Related Reading

- Release It is an excellent starting point for what defines Production Ready.

Keep Pace with Technological Change

Use the latest, most appropriate tools and technologies that solve the business problem.

Rationale

Technology advances at an ever-increasing rate. Pragmatically adopting the latest tools and technologies will allow us to provide innovative new products and services to our customers and business and compete more effectively with our competitors. Being up-to-date (but not necessarily cutting-edge) in our technology stack will help us attract and retain talented engineers.

This approach needs to be balanced against an overly diverse and complex ecosystem. A level of consistency in the technologies we use helps us develop deeper expertise, supports new engineers joining or moving between teams, and avoids us unnecessarily reinventing the wheel. Finding the "sweet spot" is a challenge.

Implications

- The principle of Evolutionary Systems is essential in order for us to update and upgrade systems without impacting the broader ecosystem.
- The best people to choose tools and technologies are those that are closest to the problem being solved. Guidelines help us adopt new technologies consistently.
- The Architecture function and the Victoria Engineering Group, supported by our engineering communities, help align technologies and identify synergies where appropriate.
- Regular patching, if possible, will help future software upgrades and maintainability and keep systems more secure.
- Conflicts in technology selection that cannot be resolved within teams should be brought to their appropriate communities or the Victoria Engineering Group for discussion, with the aim to continue advancing our technical excellence.
- We will maintain a set of favoured technology choices to help guide teams in our preferred and up-to-date choices. We encourage feedback on these choices and proposals for additions or changes to it.
- We encourage the formation and maintenance of communities of interest around the various areas of software engineering, to share good practice and technology news.
- We should look for signs that any of our existing technology choices are falling out of favour, and seek to remove our dependency on them by proactively adopting alternatives where possible, or helping direct investment in those areas where the cost and complexity of switching is prohibitive. Building Small & Simple systems makes this easier to achieve.

Model the Business Domain

Terms, concepts and capabilities of the business should be reflected in the way we write, structure and deploy our code and systems.

Rationale

Using a common language throughout the team to describe business concepts and processes allows for a common understanding. Misunderstandings that lead to defects are easier to identify because the language that is used allows non-technical stakeholders to understand and correct it.

Structuring component systems to reflect coherent business capabilities and influence the organisational structure will enable product/business owners, teams and systems to be more closely aligned to the customer value they provide, resulting in more responsive delivery and operations.

Implications

- We will need to take advantage of Conway's Law (where software reflects the organisation that built it) to validate and influence both the systems we write and how we structure the organisation.
- We will need to apply domain modelling techniques to identify bounded contexts, using a context map to align system boundaries and interfaces.
- Where packages have their own model and terminology, we may need to create abstraction layers to map to our business model.

Secure by Design

Systems will be designed and maintained with the assumption that our software and the data they hold will be attacked and possibly compromised.

Rationale

The cost of handling a security breach is significantly greater than the cost of hardening the system in the first place. There is ever increasing evidence that our customers are growing more and more concerned about their data privacy. If a breach does occur, we can minimise the scale and cost by detecting and mitigating it as soon as possible, and being proactive in identifying risks and threats to our systems.

By considering security from the beginning we limit not only the financial impact but also the significant reputational impact that would be incurred in the event of a breach.

Implications

- Teams must take responsibility for the security of their software throughout its lifecycle. This includes how they detect incidents, as well as how they respond to and resolve issues in the event of an attack or breach.
- Teams must be aware of data privacy implications including policy and legal compliance.
- Data needs to be appropriately classified and secured, both in transit and at rest.
- Teams must understand security risks and model threats to their system.
- The security of both the underlying platform used and the software being deployed to it, including any open source and 3rd party dependencies, must be considered. This applies on an ongoing basis, as threats evolve alongside the software, and needs continual review.
- People at all points of the engineering process need strong awareness of current security techniques and principles, and how to apply these appropriately throughout the lifecycle of the software.
- Teams are expected to make use of appropriate tools and techniques to identify vulnerabilities as early in the software development lifecycle as possible.

Automate by Default

Tasks that *can* be automated *should* be automated by default. The choice *not* to automate should always be a conscious decision.

Rationale

Automation of well-defined tasks, such as deployment and testing, improves speed, consistency and reliability compared to manual execution. The automation code can also be valuable as a form of documentation for the correct process or desired outcome.

Implications

- Automation requires an upfront and ongoing investment that usually pays off quickly, but must be balanced with delivering business value.
- When evaluating COTS packages or SaaS solutions, the ease with which they can be automated should be a significant factor in their selection.
- Automation may benefit from use of specialised tools, frameworks or languages, in which case teams will need to acquire or develop the appropriate knowledge & skills.
- Software and processes that are difficult to automate will need to be changed or replaced if they require too much effort or are not amenable to being automated.
- Automated tasks/processes that are complicated but infrequently executed can become a source of risk, to the extent that people decide not to perform the task manually because they don't have confidence in the automated scripts. Design for Testability must be applied so that the automation can be regularly proven to work, so that it can be relied upon when needed. Disaster Recovery processes are a good example of this.
- Automation scripts must be treated in the same way as other software and conform to these engineering principles.

Consistent Environments

Environments should have homogeneous application configuration, software, operating system, infrastructure and data (where appropriate).

Rationale

Consistency between environments reduces the time wasted investigating issues due to discrepancies, which improves the speed of delivery for the business. This consistency also helps increase our confidence in what we build and its safe delivery to our customers and end users.

Implications

- Configuration will need to be enshrined in version-controlled code.
- Software packages that are not amenable to being consistently configured will be less desirable.
- Software that is free, or provides cheaper licenses for development and testing, will be preferable.
- We can perform push-button deployments of any version of the software to any environment on demand.
- A single trunk-based CI build at the beginning of the pipeline is strongly recommended.
- We will need to invest heavily in making data available and consistent across environments. *Note: As of late November 2020, there is a lively debate going on regarding consistency of data across multiple services in nonproduction environments. There is not yet a consensus, and this implication slipped through the recent review of these principles. It should be considered "under review".*
- Creating (and destroying) new environments, including appropriate data sets, will need to be as simple and as fast as possible and will rely heavily on the principle of Automate by Default.
- Particularly where an "environment" is required to include a large set of (distributed) applications in order to facilitate end-to-end testing, the creation and maintenance of an environment can be very expensive. Reducing/limiting the number of environments may act as an enabling constraint – the fewer environments we have, the easier and cheaper it will be to keep them consistent.
- We will require consistent configuration of infrastructure (including networks, firewalls and servers).

Understandability

Each codebase must be understandable and easy to change by new developers with minimal experience of the application.

Rationale

Almost all software continues to require changes through its life, whether to fix bugs, add new features or address security vulnerabilities. Sooner or later, most software ends up being maintained by people other than those who created it. In order to be able to change the software a developer needs to be able to understand it — or at least the part of it that they need to change. Our experience is that systems typically outlast their expected lifetime, and it's not unusual for systems consigned to "maintenance mode" to be opened up for change when almost everyone who last worked on it has left the business.

Systems that are considered too difficult, too expensive or too risky to change will either become a limiting factor on business change or eventually lead to a Big Rewrite, which can cost millions and take several years.

Instead, we should ensure that applications' codebases, as well as the systems that result from the interactions between them, are easily maintainable *and kept that way*, so that they are able to change with the business requirements. Even the process of replacing or decommissioning a system will benefit greatly from its current workings being understandable.

Implications

- Software must be created with a clear set of tests which help the reader understand the application, and can be run from the command line with a single command.
- Appropriate levels of documentation will need to be created and updated, with a preference for detailed descriptions of functional behaviour to be enshrined in code as tests. This includes maintaining a README describing the application and documenting the command required to build, test and run it.
- Different audiences may require different artefacts for effective maintenance, change planning and risk management. This may include: the code, the commit history, JIRA tickets, cross-functional stories, architectural descriptions, runbooks, or formal documentation.
- Stale documentation is worse than no documentation, so investment will be required to ensure it is up to date. Maintained reference documentation and transient delivery documentation should be kept separate, so anyone can quickly understand what they can trust.
- Knowledge sharing can be split into two parts, the "WHATs" and the "WHYs". The WHATs should be evident from the code, and focused reference documentation, but the WHYs should be carefully documented as decisions are made. Decision records, code comments and/or appropriately named tests can all be useful in this case. Handover of the WHYs should be done to all levels, not just to developers.
- Each codebase should have a published set of code style guidelines, preferably importable into your IDE or text editor. At a minimum: encoding, line breaks and tabbing standards should be defined, ideally using a non-IDE-specific (or widely-supported) tool such as editorconfig.
- Developers should refactor code and tests when needed, reassured by the presence of the rapid feedback they will receive if they break something.
- An application should be consistent in applying a programming style throughout - such as procedural, functional or object-oriented. Many languages encourage a particular style, in which case this should be adopted in order to play to its strengths.

Related Reading

- On the problems with the "Big Rewrite"
  - Things You Should Never Do, Part I | Joel on Software

- Why do we fall into the rewrite trap? | Justin Fuller

Performance Importance

Our systems meet or exceed their users' expectations of performance. Degradations in performance are investigated, understood and either remedied, or accepted as appropriate in the business context.

Rationale

Systems that are fast and responsive for users enable them to carry out tasks more efficiently, and for the customer-facing areas of our business there is a strong correlation between slower websites and a negative impact on revenue and customer retention.

We must maintain a continued focus on the impact of changes to the performance of our applications or we risk impacting user experience, our ability to scale cost-effectively, and the stability of our systems.

Implications

- Set a performance budget per page or endpoint as early as possible and incorporate testing of this into the software delivery pipeline. Teams should have an awareness of how their systems have been engineered to meet performance criteria, and the means to avoid accidentally introducing changes that breach the expected user experience.
- Incorporate the analysis of performance tests into the release process. This is typically through frequent automating testing (potentially on every commit) using short component performance tests which leverage mocks/stubs. Understand what tools are available to help with the data collection and analysis of these results.
- Choose the appropriate performance technique for your workload and platform. For example, client-side performance testing tools are more appropriate for applications with a frontend. Longer "soak" tests are only required for platforms that are long-lived, and "stress" tests are more appropriate when addressing specific stability concerns or risks.
- Frequent or continuous performance testing should be implemented for the performance-sensitive components of our systems. Integration or end-to-end performance testing should be adopted very cautiously due to its high cost and complexity of maintaining fully integrated and always available testing environments - consider whether upstream dependencies can be adequately mocked under a variety of load profiles to validate any performance risks.
- Live load tests can be a useful means of validating overall end-to-end performance for our larger and more complex systems, but noting the caveats above. They should **not** be relied on to assure the performance of smaller components of the system.
- Aim to use the same tools and techniques for assuring Production Readiness when observing performance behaviour - this helps ensure that the same metrics can be relied on in the live environment, as well as engineer familiarity.
- Consider using release strategies that allow performance of significant changes to be validated using a subset of end users and rolled forward or back quickly - such as canarying or dark launching with feature toggles.

Get Feedback Early and Often

Gain feedback by frequent and early releases of functionality, rather than Big Bang releases.

Rationale

Whilst this represents a wider cultural shift, the engineering benefits are clear. Early and frequent releases through to Production provide feedback sooner, make code level integration easier, allow course corrections to be made earlier and provide business value more quickly. Problems are easier to identify and resolve in smaller changesets.

Implications

- Encourage all team members to help slice work into smaller units that can be tested and released independently.
- We encourage trunk-based development. Teams should adopt patterns such as Feature Flags and Branch by Abstraction to enable frequent deployments without resorting to long-lived branches.
- Prioritise keeping the software deployable over working on new features through constant investment.
- Requires regular assessment of feature flags and removal of the obsolete ones.
- Requires an automated, version-controlled configuration management strategy and delivery pipeline.
- No long-term development branches (and consequential complex merges) are needed.
- A common approach to controlling, measuring and evaluating outcomes of experiments is needed, particularly when systems are broken down into Small and Simple pieces.

Further Reading

- continuousdelivery.com
- trunkbaseddevelopment.com

Design for Testability

Solutions should be designed - and code structured - in a way that makes execution of its tests happen more easily and quickly.

Rationale

Solutions that are designed with testing concerns in mind facilitate faster feedback and are ultimately able to release to customers more frequently and safely. Designing for testability naturally leads to improved understandability and evolvability.

Implications

- The internal state of a component or system should be understandable through deliberately-designed external interfaces or outputs, without invasive techniques such as attaching a debugger or filling the code with debug-level logging statements.
- Structure your code to allow components to be executed in isolation in order to observe its behaviour during checking and testing.
- Ensure that the components of a system are separated into well-defined responsibilities.
- It should be easy to understand the components of a system - the code should be written in a way that is self-explaining and documented through its tests or, if necessary, through separate documentation.
- Software should be designed and structured to support automation of tests wherever possible. These tests should run quickly and throughout the development cycle to support Continuous Delivery. Use exploratory testing as a supporting testing style for non-deterministic testing.
- Carefully consider whether a diversity of technologies in use by a system introduces challenges in the testing methods and tools required, and adapt the approach accordingly.
- Release frequently through to Production. This helps build confidence in testing and reduces risk in the release process, and any issues identified soon after the cause are easier to fix whilst still fresh in the engineer's mind.

Further Reading

- Testability blog post by Michael Bolton.
- Team Guide to Software Testability by Ash Winter and Rob Meaney.