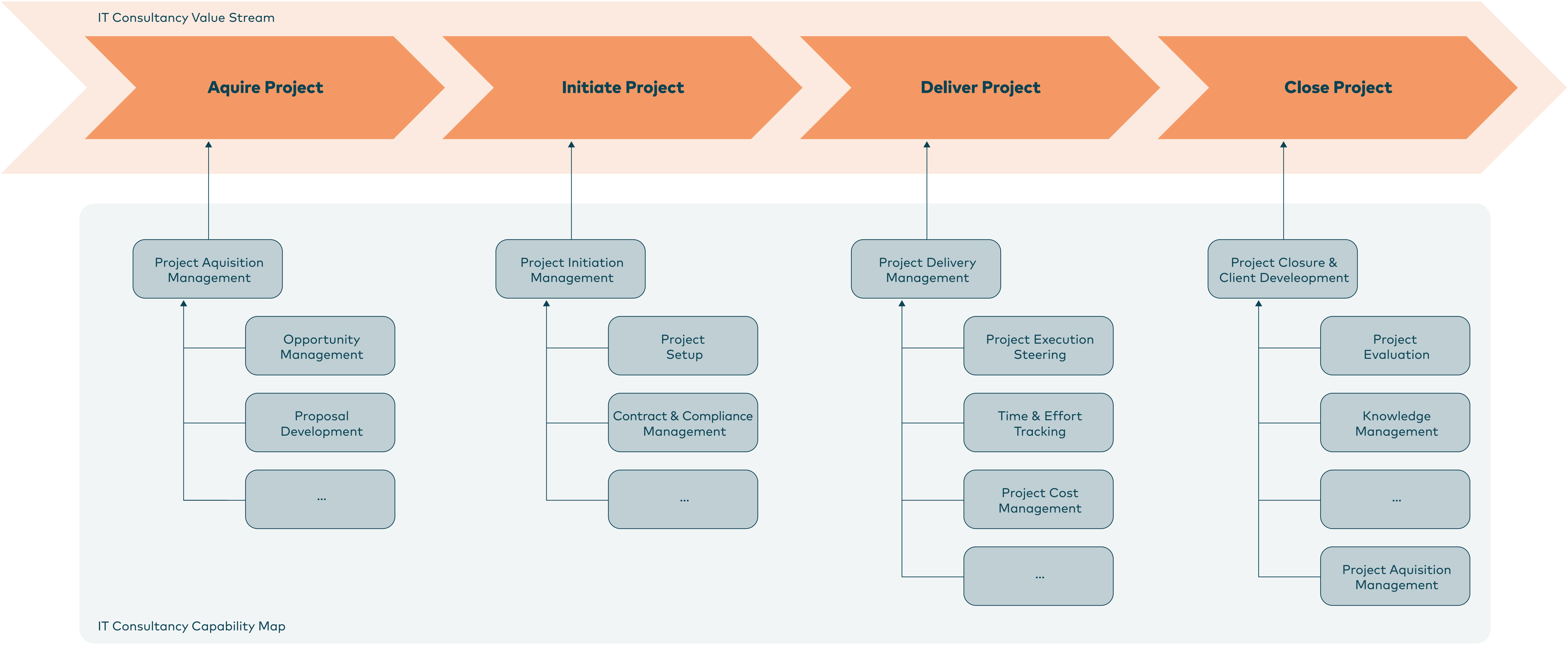


The figure "Capability Map" shows (abbreviated) how a Capability Map could look in detail for an IT consulting firm (based on ArchiMate). Along the defined Value Stream[17] from "Acquire Project" to "Close Project," the required capabilities are arranged hierarchically. This makes it easy to see which capabilities are relevant across entire value creation process.

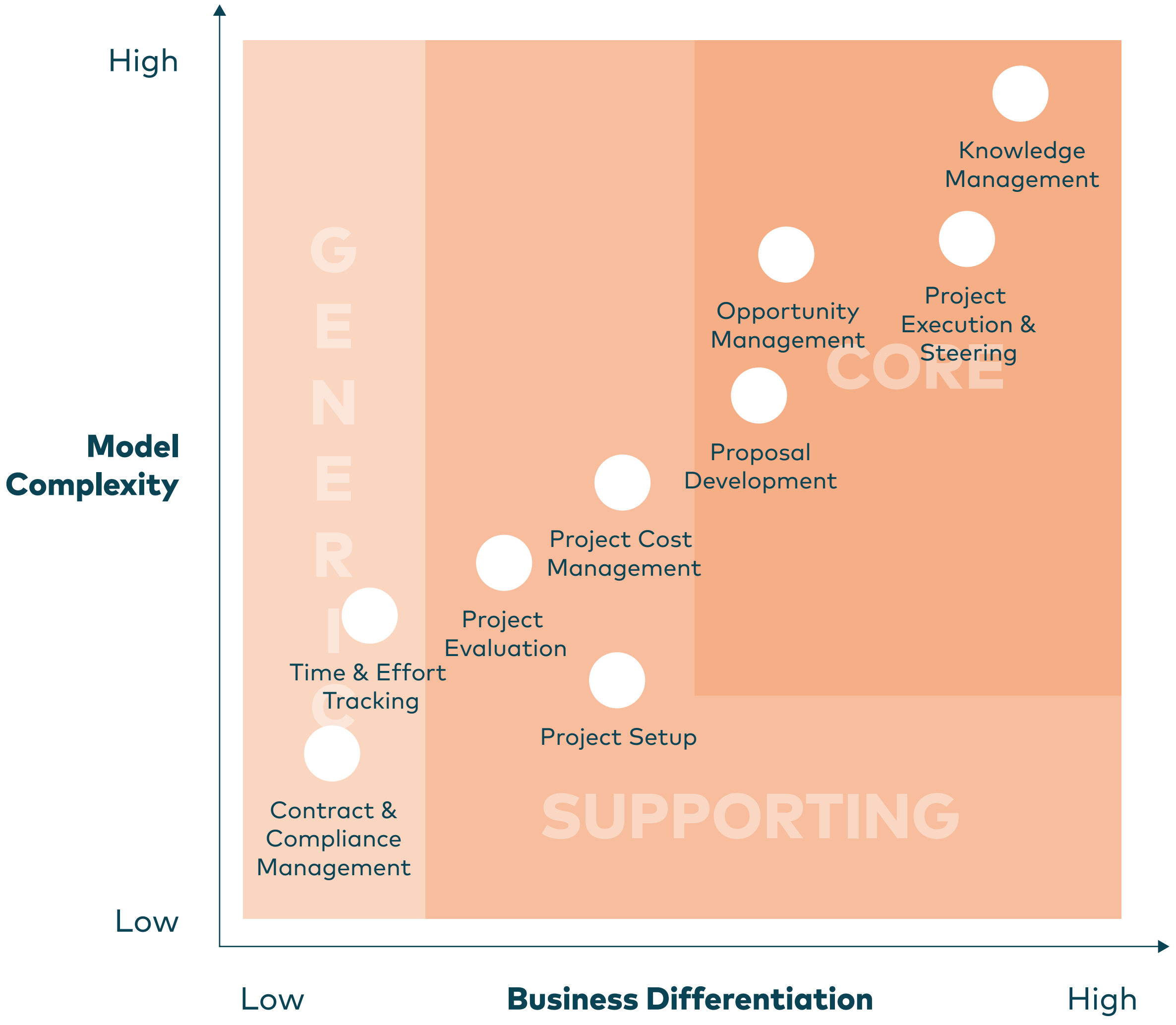
The evaluation of these capabilities is done using proven methods such as Wardley Maps[23], DDD "Core-Domain-Charts"[24] or strategic workshops. It's important to reflect on the maturity level, strategic importance, and desired market differentiation of each capability.

This allows you to identify three groups:

Core Capabilities: Areas where we must be better than the market – here, custom development is worth the investment.

Supporting Capabilities: Supporting functions that are important but not differentiating – standard software is often the best fit here.

Commodity Capabilities: Interchangeable areas where efficiency is the priority – standard software is the preferred approach here.



In our IT consulting example, the capability "Time & Effort Tracking" is classified as *Generic*, while "Project Execution & Steering" is considered *Core* because it directly influences value creation in customer projects. "Time & Effort Tracking," however, follows largely standardized processes and offers little differentiation potential, making standard software standard software the logical choice here.

This analysis results in a make-or-buy strategy that determines for each capability whether it should be developed internally or handled by standard solutions. A color-coded Capability Map (e.g., blue for custom development, green for standard software) creates transparency and serves as a central management tool for further planning.

This creates an IT landscape where standard software is purposefully deployed where it creates freedom – and custom development takes place where it delivers the greatest added value. This is the first step toward using standard software not randomly, but deliberately and with sovereignly.

Architectural Guardrails for Greater Freedom

The decision of where to deploy standard software is only the first part. Equally important is how this standard software is integrated. Without clear architectural specifications, even the smartest make-or-buy strategy can quickly result in a confusing, difficult-to-change patchwork.

This is why a macro-architecturer[15] is needed: overarching guidelines that apply to all systems – regardless of whether they're custom developments or standard products. This architecture limits itself to a manageable number of central topic areas while ensures that there are common standards at the crucial interfaces. This creates homogeneity within a heterogeneous system-of-systems: individual systems can be developed or replaced independently without destabilizing the overall landscape.

This is especially essential when deploying standard software. Integration guidelines must ensure that coupling between systems remains as loose as possible to enable vendor switches when needed without major effort. In practice, asynchronous, event-driven architectures[25] have proven effective patterns because they decouple systems and enable flexible responses to changes. Accordingly, companies should ensure that, when selecting standard software, it includes suitable APIs and ideally already publishes events independently – for example, via webhooks.

A common mistake is directly forwarding proprietary data structures to other systems. This creates dependencies that are difficult to resolve, making later vendor switches extremely expensive and risky. Instead, incoming data should be transformed into company-specific formats via wrappers. While this initially increases integration effort, it reduces long-term complexityand facilitates reusability.

Finally, when selecting central integration components – for example, the event broker – it's important to ensure they're based on open and standardized formats. Proprietary protocols at this critical point would again create a dependency for the entire system that is difficult to remove. An open, ideally open-source-based approach protects long-term independence.